# BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

## IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.**

UNITED STATES PROVISIONAL PATENT APPLICATION

FOR

# KINZAN ARCHITECTURE AND TECHNOLOGY PLATFORM

Inventors:
Garland Wong
Carlos Chue
Anthony Tang
Reza Ghanbari
Dyami Calire
Eric VanLydegraf
Meliza P. Sanchez
Trent Barnes

Prepared by:
BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard
Seventh Floor
Los Angeles, California 90025-1026
(408) 720-8598

Attorney's Docket No: 4348P006Z

"Express Mail" mailing label number: EL627471455US
Date of Deposit: 1st February 8 2001
I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Assistant Commissioner for Patents, Washington, D. C. 20231

GLENN E. VON TERSCH
(Typed or printed name of person mailing paper or fee)

Glenn E. Von Tersch
(Signature of person mailing paper or fee)

February 8, 2001
(Date signed)

# KINZAN ARCHITECTURE AND TECHNOLOGY PLATFORM



In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one skilled in the art that the invention can be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to avoid obscuring the invention.

Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment, nor are separate or alternative embodiments mutually exclusive of other embodiments. Moreover, various features are described which may be exhibited by some embodiments and not by others. Similarly, various requirements are described which may be requirements for some embodiments but not other embodiments.


# Technology Platform Developer's Guide

**Kinzan, Inc.**
2111 Palomar Airport Road
Carlsbad, California 92009



4348P006Z

# KTP Developer's Guide

# 1. Introduction

The goal of the Technical Preview 4 release of the Kinzan Technology Platform (KTP) is to provide web-based applications that are easily assembled and extended by developers and development partners with a wide variety of skills.

The various components of the KTP Framework are intended to untangle the web of static and active content, style, structure, application logic, templates, servlets, and persistence mechanisms that is the norm in application environments that are driven by Java Server Pages (JSP). The result represents a major commitment to flexibility, configurability, extensibility, integrability, scalability, and reliability at all levels of application development.

## 1.1 Assembling Is Easier than Building

The KTP Framework enables software engineers to work collaboratively with information architects, web developers, graphics designers, and customer advocates to create compelling network-based applications that can be delivered to different types of network devices (i.e., web phones, browsers, etc.), with different branding, localized to different languages.

The KTP Framework offers a series of abstractions that modularize and generalize development at all levels of the application architecture. The result is an environment in which applications can be assembled from Adaptive Web Service (pre-built modules), minimizing the amount of custom work and maintenance required for individual customers.

At the front end, the primary mechanism for assembling a web page is a Kinzan Page Descriptor (KPD), which is actually a collection of configuration and content files. The rendering process, which involves recursive assembly, transforms a given page into a series of precompiled Java servlet modules, allowing developers to develop collaboratively and maintain modules at a level that is appropriate to their role and skill set.

At the back end, the Kinzan Services Manager provides a powerful and flexible mechanism for integrating and coordinating a large variety of back-end databases, information sources, enterprise applications, and services.

Bridging the front and back ends is an extremely rich environment for assembling application logic and flow from application modules, supported by the Kinzan State Manager.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

## 1.2 Make World-Class Technologies Accessible

At all levels the KTP Framework leverages the best that the Java 2 Enterprise Edition (J2EE) platform and XML-based technologies have to offer, and takes those technologies to a new level. The result is a highly scalable, reliable, extensible, and distributable platform that radically decreases development time.

By modularizing and generalizing much of server-based application development, the KTP Framework facilitates development of complex enterprise-hosted applications, enabling fully branded, highly integrated customer experiences.

Perhaps more importantly, the modularization and significant simplification of the core J2EE technologies allows highly productive and impactful contribution by development team members with a wide variety of technical and design skills. Each can focus on modules where their skills have the greatest impact. Extensive modularization also encourages reuse and sharing of modules, increasing productivity and quality in the process.

## 1.3 End-to-End Support

The KTP Framework is more than just software. It has been designed to integrate seamlessly with a high-availability, high-scalability application-hosting environment. The result is a technology platform that enables developers to create compelling applications rapidly, and to deploy those applications rapidly and painlessly to a mission-critical hosting environment that can grow as demand for the applications grow.

## 1.4 One Platform, One Web

The KTP Framework has been built to bridge a myriad of back-end services and data sources to multiple target devices (web phones, browsers, PDA's, etc.) with multiple branding and, potentially, to multiple languages and locale.

Services that are integrated on the back end are made available to application developers as Kinzan Widgets. Widgets support unique presentation and branding to multiple target devices and locales, allowing the web developer to take advantage of the rich functionality the widget exposes. In turn, Widgets are bound to business rules and application specific data to form Components. Components are then bundled with XML descriptor files to become Adaptive Web Services, which can be downloaded and integrated into any application that is leveraging the KTP Framework.

As Kinzan and its partners integrate services such as e-mail and payment processing into the KTP Framework, the services become available to all




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

applications built with the KTP Framework as their base. The result is a progressively growing set of services and widget components that can be assembled into adaptive web services and incorporated into compelling, branded applications.

# 2. Motivation

Fundamentally, the KTP Framework is intended to enable rapid branding, assembly, extension, and configuration of generic application modules. It does so by modularizing style, presentation, content, application logic, business logic, and services, and providing runtime capabilities that dynamically assemble these modules into rich applications.

The result is a development and deployment environment that cleanly separates the various elements required to build dynamic web applications. Separating these elements eliminates the need for all team members to have multiple skill sets (i.e., graphics design, page layout, Java programming, etc.). Teams work more effectively, with various team members contributing to those modules that require their skills

## 2.1 Market Drivers

Current server-based application development technologies tightly couple presentation, content, and logic, making the modular development of brandable applications impractical. Each application becomes a one-time implementation, compounding support and maintenance issues going forward.

Kinzan has a long history of delivering uniquely branded and configured Internet content and commerce-management systems to our customers on our hosted platform. The result is a deep understanding of application frameworks that enable the rapid branding and configuring of generic applications.

Competitive pressures mean time to value delivery is key. This places a premium on being able to rapidly and easily assemble, reconfigure, and extend generic applications. The global nature of the Internet also places a premium on easy localization and support for emerging alternative devices for Internet access (PDAs, phones, pagers, etc.).

Finally, supporting an application development and deployment environment in a shared hosting facility requires robust security and reliability to support multiple development partners simultaneously.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

## 2.2 Technology Drivers

Even the most sophisticated technologies have limited value if they are too complex to use. There is a critical need to develop and package the most sophisticated technologies in a way that makes them accessible to teams with varying technical skills.

Typical client-server implementations separate the client from the database. In the JSP world, that means direct connectivity between the JSP and the database via Java Database Connectivity (JDBC) queries (so-called Java Model 1 architecture).

With Enterprise Java Beans (EJBs), developers may abstract out business logic into a separate services layer using entity EJBs and session EJBs. However, that still leaves presentation and application logic in the JSP layer. In JSP 1.1, using tag libraries helps to extract the application and presentation logic associated with common components out of the JSP; but the different elements are still co-mingled. Combining JSPs and servlets in a Model-View-Controller (MVC) configuration (so-called Java Model 2 architecture) alleviates some issues, but generally requires a higher level of sophistication for all developers on a project.

The KTP Framework (including rendering, state management, and services management processes) cleanly separates style, layout, presentation, and content from application and business logic. It also supports localization and output to target deployment platforms with different capabilities (PDAs, browsers, pagers, etc.).

The collection of files that make up a KSP are the "source code" that is "compiled" into a series of modular servlets (and associated data), which are then dynamically assembled by the rendering engine at runtime. In this way, developers can take full advantage of the power and performance of Java servlets, while benefiting from increased reuse and modularity and the clean separation of function and responsibilities in the development process.

The Kinzan State Manager uses XML-based wizards to connect and configure application logic modules, effectively assembling the controller tier of the application.

The Kinzan Services Manager provides an abstraction layer for many kinds of back-end services and includes the ability to dynamically look up and bind to services. The result is modularization of the model tier.

The KTP Framework is flexible. If developers choose to implement their pages "close to the metal" as standard JSP pages or Java servlets, they may. However, to do so is to give up the benefits of being able to apply different style and branding elements to a given application. In practice, developers will likely leverage the KTP Framework for elements of their applications that require flexible styling and branding and apply conventional HTML, XML, and JSP techniques for sections where the flexibility is not required.



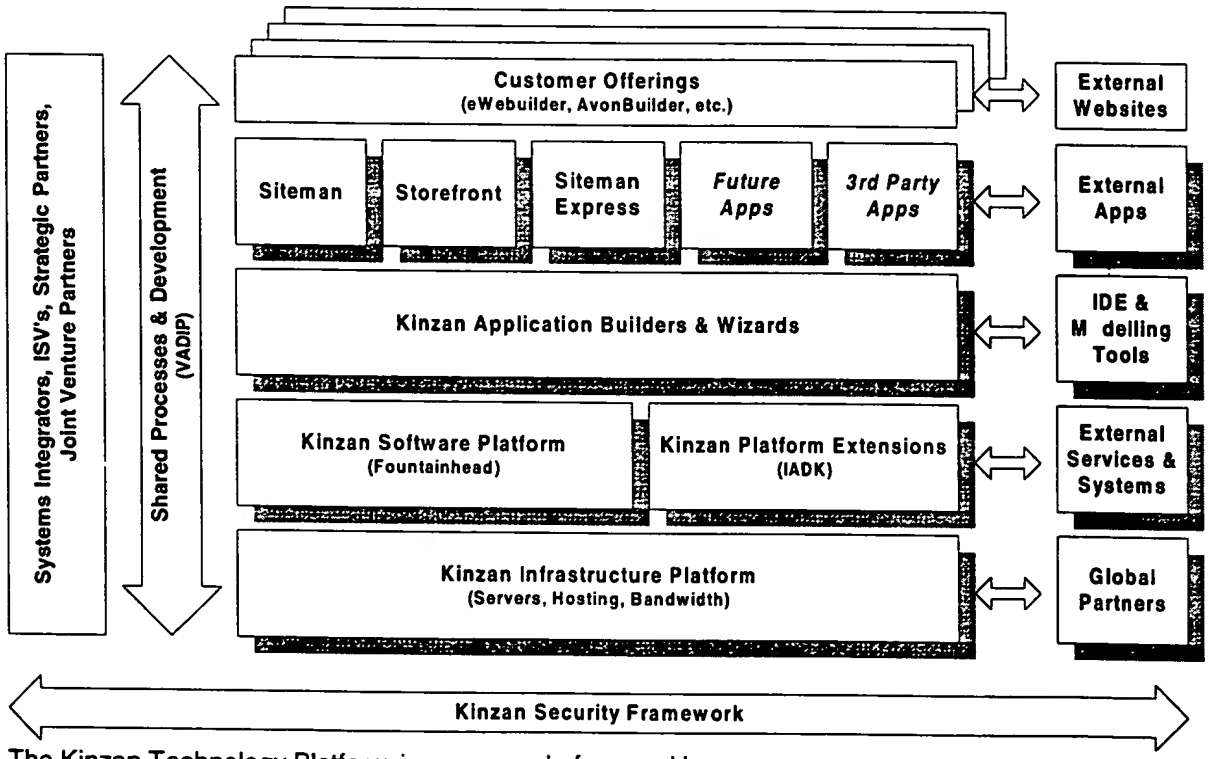


2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

The KTP Framework is designed to support the development and deployment of closely related applications across multiple capability devices (pagers, PDAs, etc.) and locales (English, French, etc.).

The KTP Framework allows developers to rapidly develop and deploy closely related applications to platforms with different capabilities, such as a standard web browser and a two-way pager with a 20 character display for example. By only needing to change the presentation modules, developers can leverage a common development methodology, business logic, localization, and data persistence layers between all apps.

# 3. Kinzan Technology Platform Overview




The Kinzan Technology Platform is composed of several layers.

At the base is the Kinzan Infrastructure Platform. This represents the high-availability, high-scalability hosting infrastructure for the KTP. These data centers can be shared among many partners and customers, giving each access to more capacity and more reliability at less cost.

The Kinzan Software Platform (also know as Fountainhead) represents the core services and capabilities described in this document. It is a shared foundation for all applications built on top of the KTP.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

The core platform can also be extended by integrating in new services, then exposing these services with widgets and wizards.

The modular nature of the KTP encourages reuse at all levels. Collections of modules (widgets, wizards, and support modules) form libraries that may be used to streamline application development.

Kinzan's own generic content-management and commerce-management applications are assembled from these modules. In turn, these generic capabilities are assembled and configured into uniquely branded offerings for each customer.

Kinzan's Virtual Application Development and Integration Process (VADIP) ties all these pieces together, while the Kinzan Security Framework offers enhanced information security above and beyond what is typically found in the Java platform.

## 3.1 Empowering Project Teams

At all levels, the KTP Framework works to make technology accessible to teams with different skill levels. Specifically, it provides a full dynamic Rendering Service that assembles various applications on the fly. A benefit of this approach is the ability for applications to directly configure the runtime environment of other applications, effectively resulting in codeless development for those interested in maintaining their own applications.

For web developers, the KTP Framework exposes powerful J2EE features as HTML-like tags for advanced components (Kinzan Widgets), allowing web developers to focus on creating new and better presentations for the generic functionality in these widgets.

For graphic designers, the KTP Framework supports a Cascading Style Sheets (CSS) -like approach to styling generated output. However, unlike traditional CSS, which requires the browser client to interpret the CSS, the KTP Framework applies the CSS transformations on the server-side. The result is nearly all of the benefit of CSS, without the client-compatibility issues.

In addition, the KTP Framework provides an easy way to manage assets (images, documents, etc.), generically managing asset variants based on any combination of style, locale, and target device. For example, a logo asset may include a large JPEG variant for web pages, a small BMP variant for web phones, multiple color variants to compliment different corporate color schemes, and perhaps a localized slogan for different regions of the world.

The graphic designer manages all these variants, and the rest of the team automatically inherits these variants whenever they use the logo asset. As the asset is enriched with new variants, the Kinzan Rendering Service always



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

resolves the variant that is most appropriate to the user who requests a page from the web application.

For more advanced web developers, the KTP Framework supports an enhanced version of Java Server Pages that is used to define in-line logic and presentation for components (widgets). Widget presentation is managed as an asset, so widgets may have multiple variants based on style, locale, and/or target device. Managing presentation for widgets is the bridge between web development and software engineering, and leverages the strengths of web developers who have acquired some sophistication with JSP programming.

For mid-level software engineers, the widget component model for presentation and the pipeline component model for application logic enable development of cleanly encapsulated components that can be easily configured and connected to each other using XML.

For more advanced software engineers, the Kinzan Services Manager enables sophisticated integration and development of back-end services and components, including Enterprise Java Beans for business logic and integration with existing enterprise systems. These services may then be exposed with widgets and rolled out to many applications.

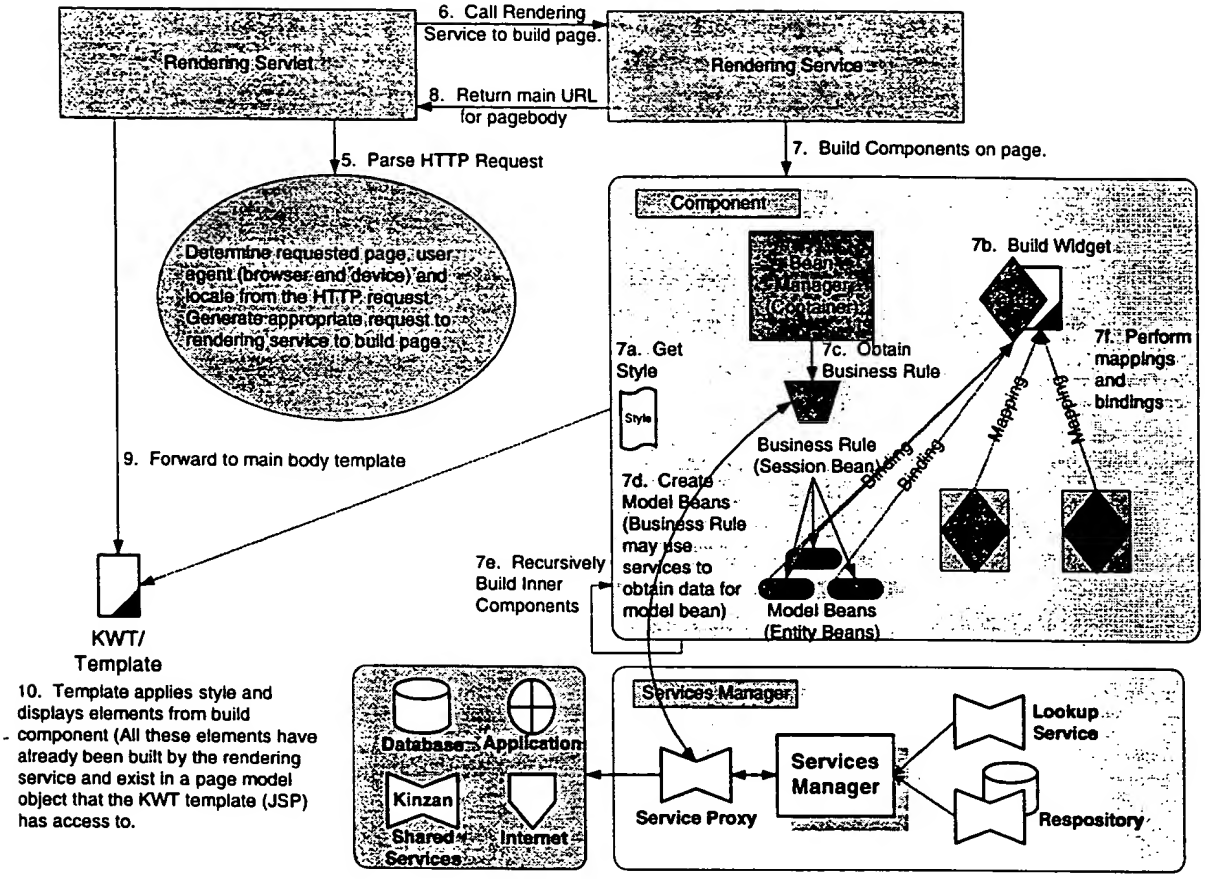
At all levels of the KTP Framework, support for dynamic assembly of modules enables the easy reuse of modules with resource libraries (pages, components, styles, structures, etc.). The result is more effective development, less maintenance and support, and higher quality systems.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 3.2 Modular Assembly Approach




The KSP Framework offers a modular assembly approach to page and application definition. By leveraging real-time modular assembly of pages by the Rendering System, and modular assembly and control of applications by the Kinzan State Manager, KTP-based applications are intrinsically extremely flexible and configurable.

The modular assembly approach also empowers less technically sophisticated team members to assemble sophisticated applications, rather than having to code the various elements themselves. More importantly, it makes it practical to develop tools and applications that directly manipulate and configure other applications, empowering customers to maintain their own applications via "codeless" development.

The KSP Framework decomposes style, structure, presentation, active content, branded assets, application logic, business logic, and various support services into distinct modules. Development team members contribute by developing





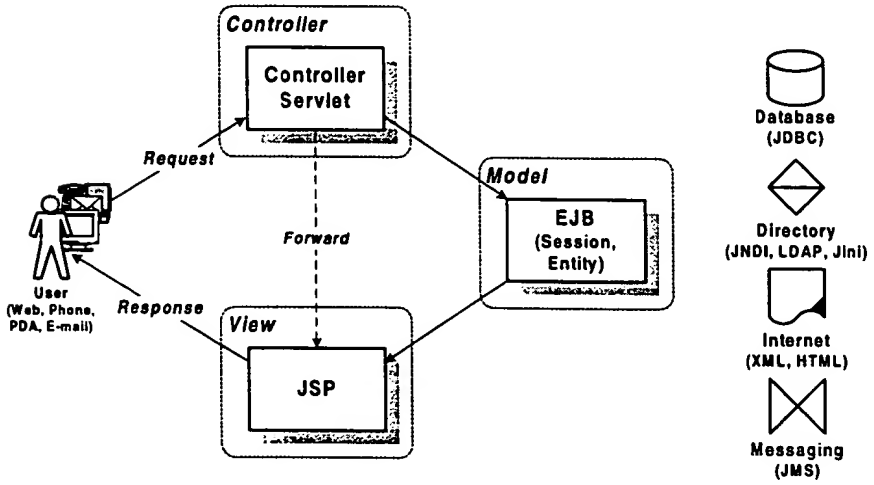
2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

modules that are appropriate to their skill set. These modules are then assembled to describe web pages and web applications.

The KPD is the artifact that describes the modules to be used to assemble a page, and the Kinzan State Diagram is the artifact that describes how pages and application logic are to be assembled into dynamic web applications.

Although the KTP Framework attempts to isolate dependencies between the various element types described in the Appendix, there are situations where rules are necessary to control the coupling between element types. Examples are included in the next section. In these cases, the KTP Framework allows definition of module variants that are bound to any combination of style, locale, and/or target device. Changing any of these instantly changes the module variant that is used when rendering a page or driving the application.

## 3.3 Request-Response Architecture




Request-Response: Java Model 2 Architecture

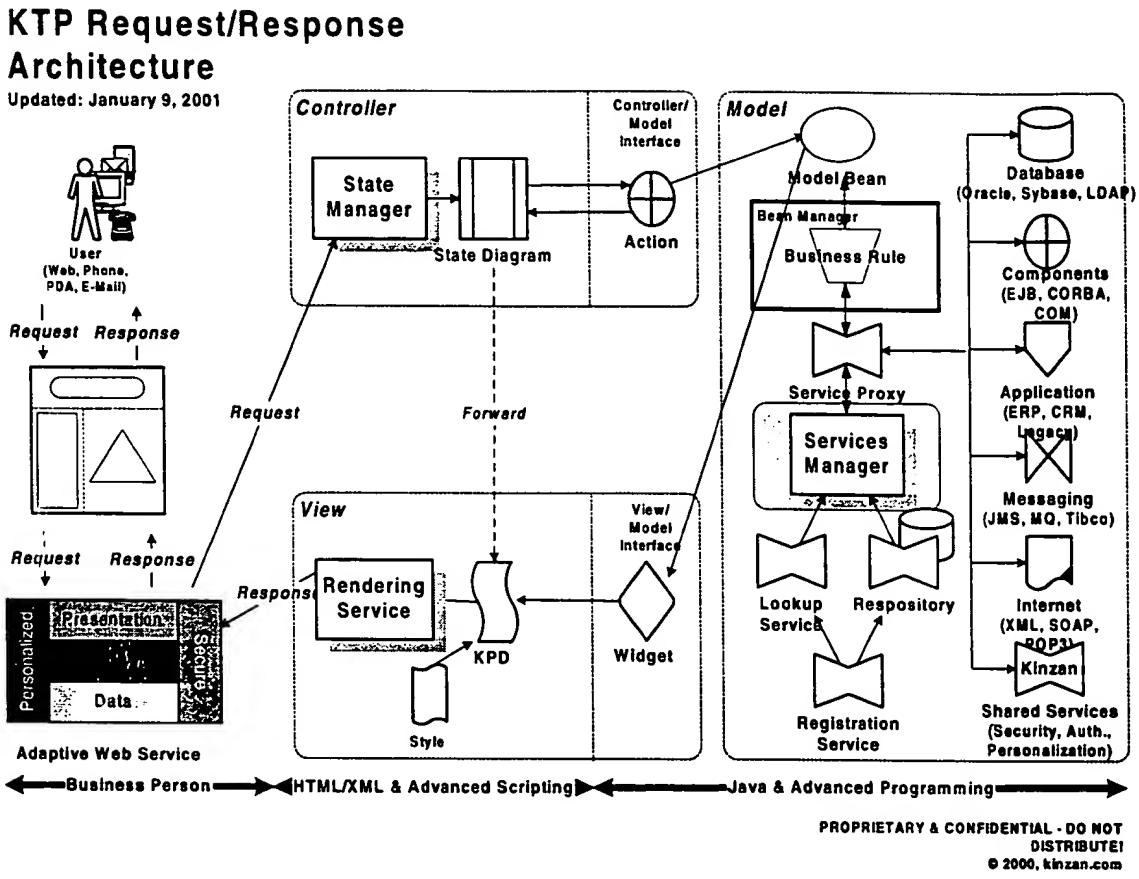
In the Java Model 2 architecture, Java Server Pages (JSPs) are used to query Enterprise Java Beans (EJBs) to generate a dynamic web page for a user. Different JSPs may be written to output different XML variants to support different kinds of devices.

HTML is embedded in the JSP, along with the Java programming that is required to access the EJBs. Since sophisticated developers and sophisticated web designers are rarely the same person, it is often difficult to support a collaborative development environment.

When a user performs an action (say, clicking the check out button at an e-commerce site), the web page passes the action to a Java controller servlet. This servlet processes the action, stores information in EJBs if necessary, then asks that the next appropriate JSP get rendered.




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN




Request-Response: Kinzan Technology Platform

The KTP Framework takes each of these layers and modularizes it. At the view layer, the Rendering Service dynamically assembles pages from all the modules that make up the page. Interfacing with the model layer is managed through widgets, minimizing the impact on those that are more expert with graphics design and page presentation.

The State Manager performs a similar function at the controller layer. Kinzan State Diagrams are used to dynamically assemble and configure application-logic modules called model beans. Interfacing with the model layer is managed with model beans, allowing information architects and site designers to more easily manage the configuration of application flow.

The Services Manager provides an abstraction and resolution layer on top of many kinds of services and components, making it easier for widget and application-logic developers to take advantage of rich back-end services at the model layer.

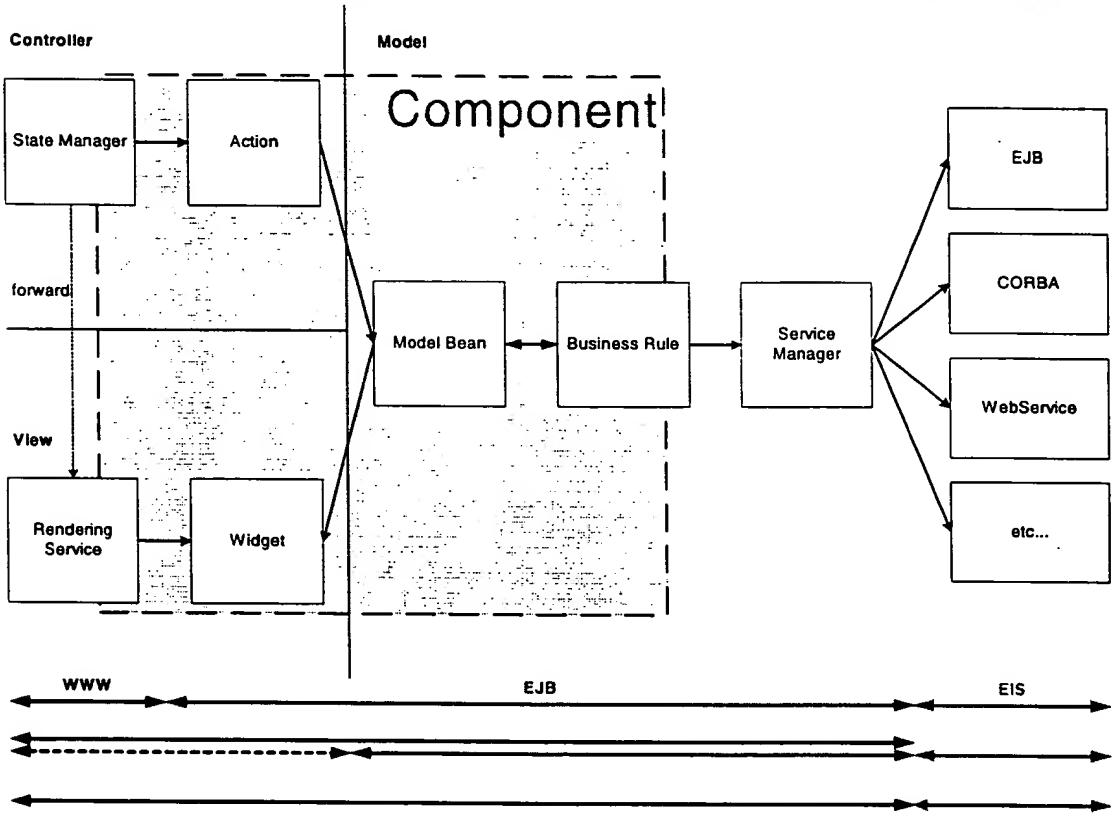




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

# 4. The Component Overview

## 4.1 KTP Component Overview




KTP Component Overview

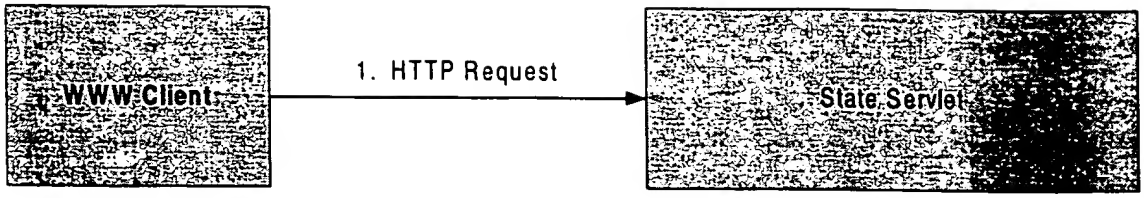
The Kinzan Technology Platform component architecture is comprised of the following: view, controller, and model. This design enables all aspects of a component to be packaged so that they can be deployed as services and downloaded from the Internet as Adaptive web services.

## 4.2 Component the Request





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

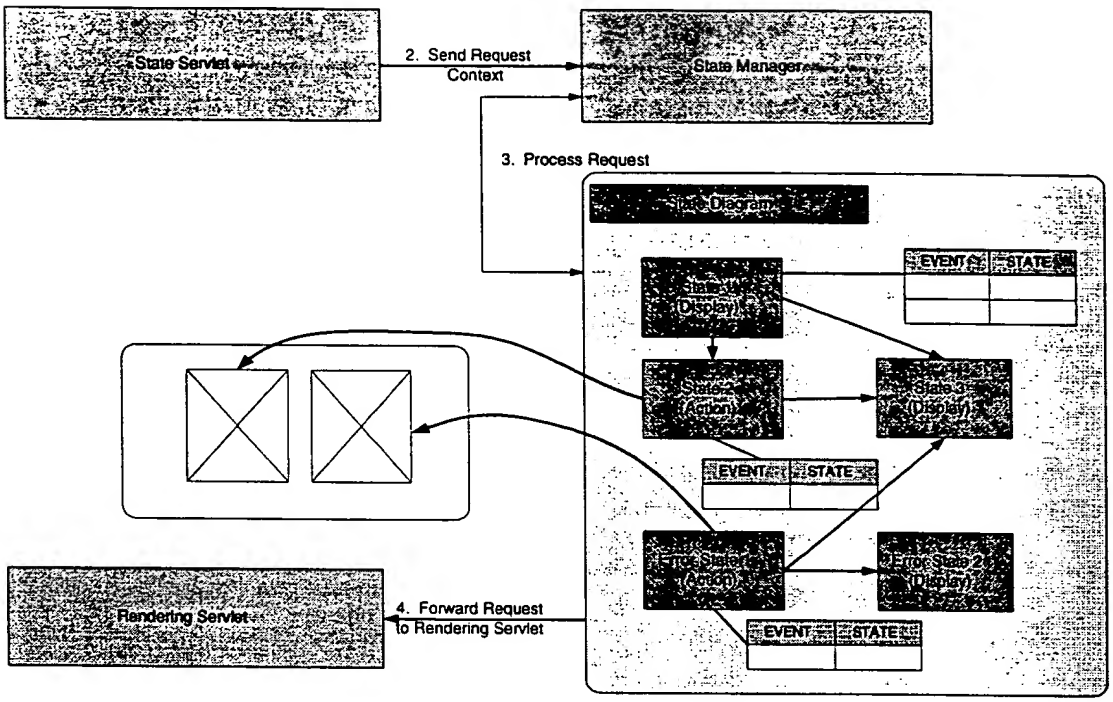



The HTTP Request is in the form:
http(s)://address:port/stateServletName/application/ksd/state?requestContextID=RQID&EVENT=event
where
stateServletName = alias for state servlet in the servlet runner
application = name of the application/site that the state diagram exists in*
ksd = name of the state diagram to run*
state = state in the state diagram to run*
RQID = Identifier for the current request context/session*
event = The name of the event to post.
* Typically, an application that is in process will provide a requestContextID, which automatically identifies the application, ksd, and state to the state manager. The ksd also contains a default state so that state does not need to be specified. The request context ID and event name are the keys which allow the state manager to identify the next state to execute or display.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 4.3 Component Processing the Request




## 4.3.1 State Diagram Detail

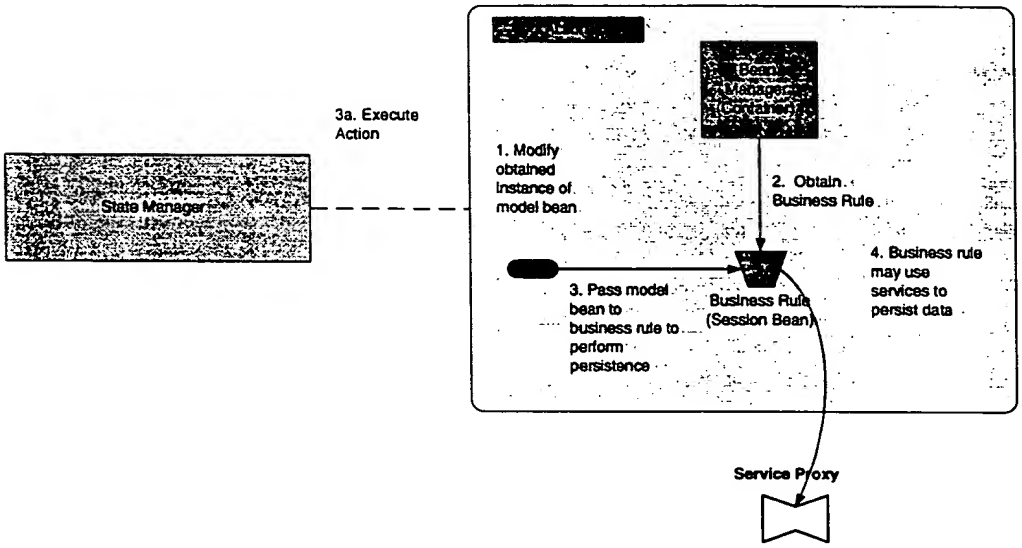
```
<ktp:wizard name="LoginWizard" startState="Login" directAccess="false" visibility="public">

    <ktp:displayState name="State1" kpd="KPD1" directAccess="true" >
        <ktp:transition event="Event1"  state="State2" />
        <ktp:transition event="Event2" state="State3" />
    </ktp:displayState>

    <ktp:actionState name="State2" component="net.kinzan.component.Component1">
        <ktp:transition event="Event3" state="State3" />
    </ktp:actionState>

    <ktp:displayState name="State3" kpd="KPD2" />

    <ktp:actionState name="failure" component="net.kinzan.component.Component1">
        <ktp:transition event="Event4"  state="State3" />
        <ktp:transition event="Event5" state="failure2" />
    </ktp:actionState>

    <ktp:displayState name="failure2" ksp="defaults/failure"/>

</ktp:wizard>
```





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 4.4 Component Performing an Action

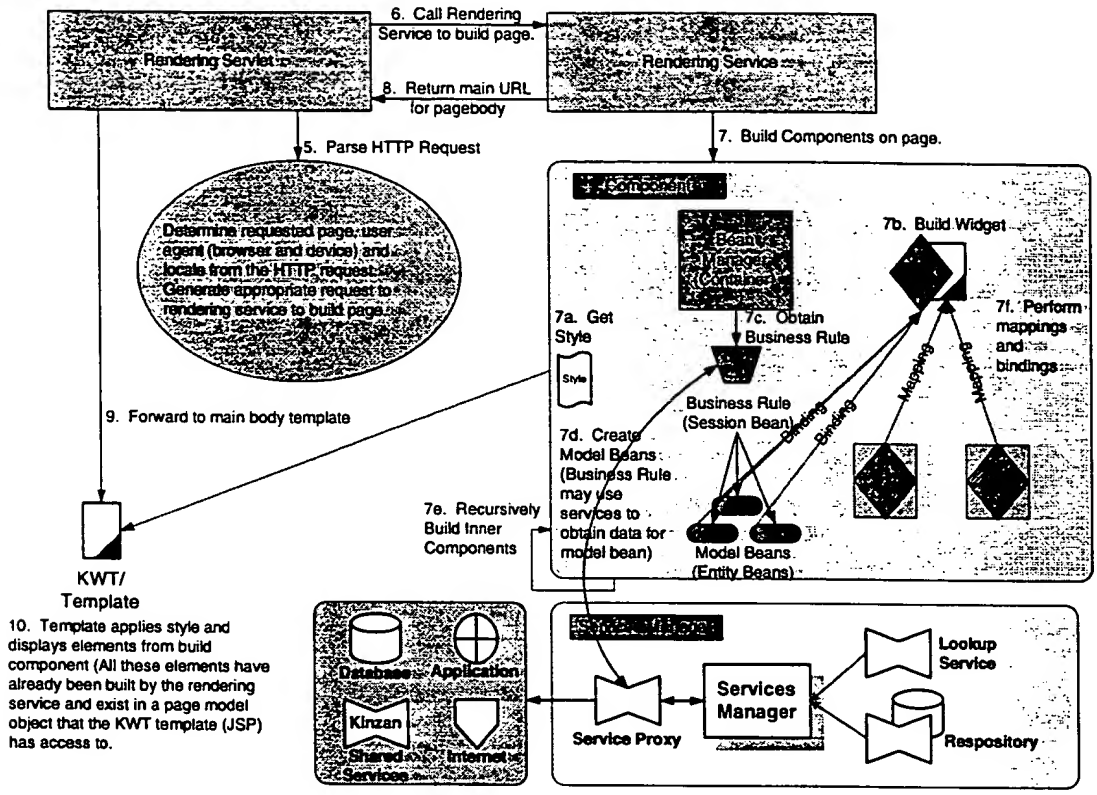







2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

# 4.5 Rendering the Component




## 4.5.1 The Widget

```
<ktp:widget name="sampleWidget">
    <!-- The attribute list defines bindable attributes in the widget. A class
         is automatically be generated from the attribute list. -->
        <ktp:attributeList>
            <ktp:local name="text" type="java.util.String"/>
        </ktp:attributeList>

        <!-- style, device, and locale attributes default true and indicate
             to the preprocessor whether such variants should be searched for -->
        <ktp:widgetTemplate>sampleTemplate.kwt</widgetTemplate>
</ktp:widget>
```

## 4.5.2 The Widget Template





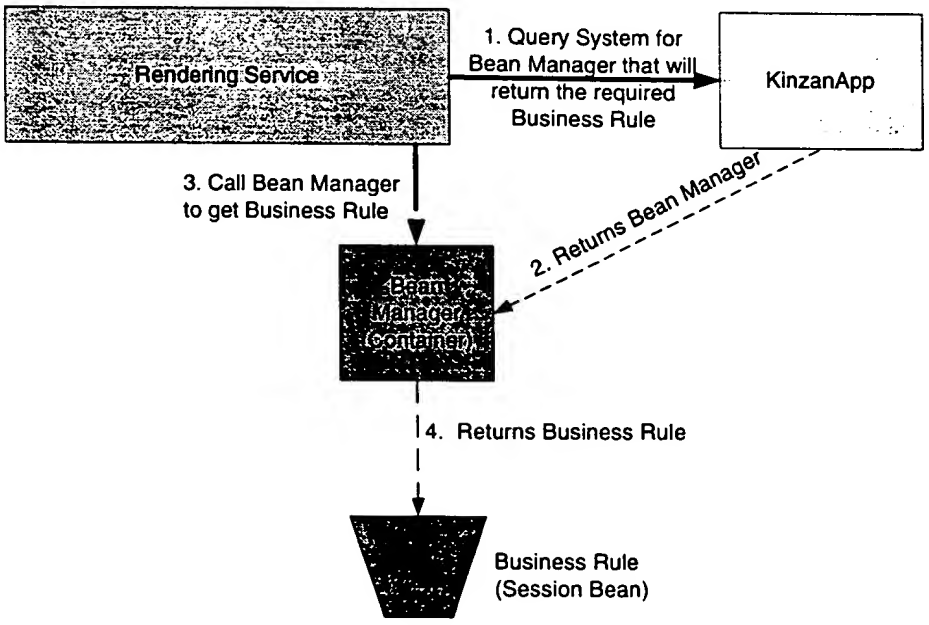
2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

```
<!- Sample KWT file for simple widget example -->
<table>
    <tr><td><ktp:zone name="zone1"/></td></tr>
    <tr><td><ktp:attribute name="text"/></td></tr>
</table>
```

• **Zone** is a placeholder for other components to be **mapped** into

•**Attribute** places the value of the named attribute from the widget into the template.The loader processes the KWT into a JSP containing the appropriate JSP code to perform the required mappings.

## 4.5.3 The Bean Manager



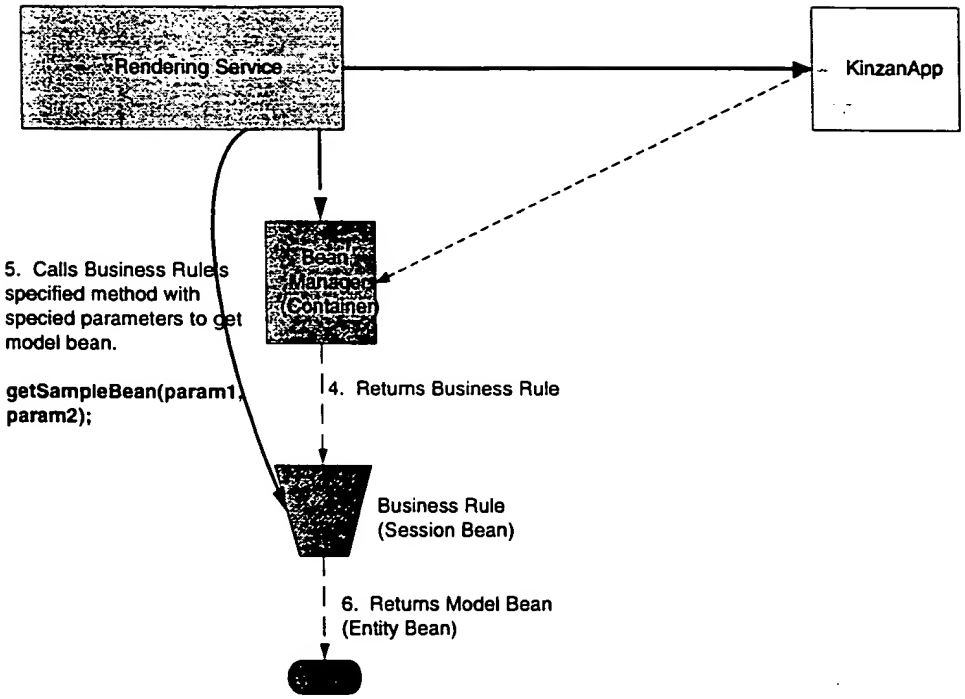

```
<ktp:beanManager name="SampleBeanManager">
    <ktp:beanClass>net.kinzan.model.beanmanager.SampleBeanManager</ktp:beanClass>
    <ktp:propertyFile>
        http://codebase.kinzan.com/codebase.SampleBeanManager.properties
    </ktp:propertyFile>
</ktp:beanManager>
```





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T. 760.602.2900 F. 760.602.2910 www.kinzan.com KINZAN

## 4.5.4 The Business Rule (1)




```
<ktp:modelBean name="sampleBean">
    <ktp:rule name="sampleBusinessRule" method="getSampleBean"/>
    <ktp:param name="param1" type="java.lang.String">
        <ktp:from widget="sampleWidget" attribute="text"/>
    </ktp:param>
    <ktp:param name="param2" type="java.lang.String">
        <ktp:from literal="param2"/>
    </ktp:param>
</ktp:modelBean>
```

## 4.5.5 The Business Rule(2)

```
public class SampleBusinessRule extends KinzanBusinessRule
{
    ...
    public SampleBean getSampleBean( java.lang.String param1, java.lang.String param2 )
    {
        InitialContext ic = new InitialContext();
         // Get home interface
        SampleBeanHome sbh = (SampleBeanHome)ic.lookup("SampleBeanHome");
        // Get remote interface to entity bean.  param1, param2 form the primary key.
        SampleBeanPK pk = new SampleBeanPK();
        pk.key1 = param1;
        pk.key2 = param2;
        SampleBean sb = sbh.findByPrimaryKey( pk );
        return sb;
```

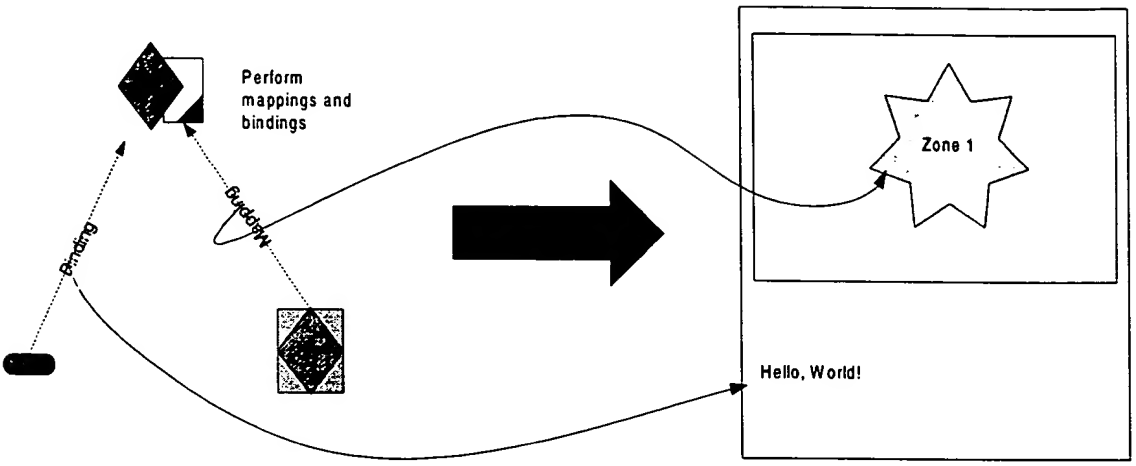




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

```
    }
    ...
}
```

### 4.5.6 Binding and Mappings




```
-
    <ktp:binding widget="SampleWidget" attribute="text">
        <ktp:from modelBean="sampleBean" attribute="message"/>
    </ktp:binding>
-
<ktp:componentModeList>
    <ktp:componentMode widget="sampleWidget">
        <ktp:zone name="zone1" component="anotherComponent"/>
    </ktp:componentMode>
</ktp:componentModeList>
```

# 5. The Kinzan Application

## 5.1 The Kinzan Application Hierarchy

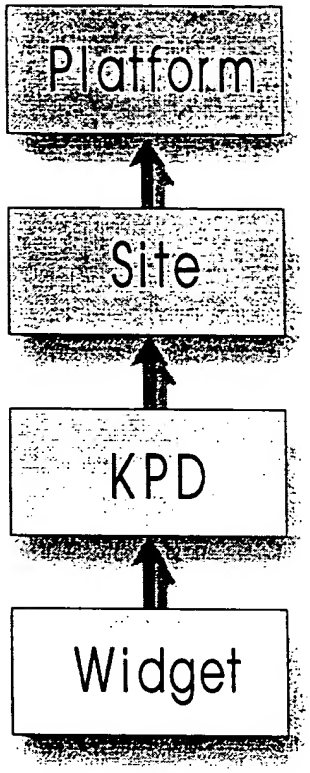
Within the Kinzan Technology Platform, sites are arranged hierarchically. Resources may be inherited or overridden at each level.

Each Kinzan Web site is actually part of a large hierarchy of sites. Resources associated with sites at upper levels are available to sites at all lower levels. Lower levels may provide their own variant for a resource, overriding the more global declaration. This hierarchical arrangement allows you to build libraries of



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

resources, streamlining development and facilitating the reuse of common modules.




Generic resources visible to all applications and pages developed on the platform.

Changes here are automatically and immediately applied throughout the hierarchy unless the resource has been overridden at a lower level.

Resources associated with a particular site or application—template bundles that represent all the brandable resources in an application. Sites may also be bound to parent sites; all resources available to parent sites are recursively available to child sites.

Resources required to render a particular page, including other KPDs (if necessary). Changes here occur only on the specific page

Resources specific to a particular widget instance. This is the termination point of the Kinzan hierarchy.

Resources at a lower level override identically named resources declared at a higher level. For example, if the platform provides a generic widget named *stockQuote*, and you want to replace it with a different implementation, you need only declare a new widget named *stockQuote* at a lower level in the hierarchy. Although you may also implement a different widget, one named *myStockQuote* for example, using the new widget requires you to modify all references to *stockQuote* within the various KPDs in the application.

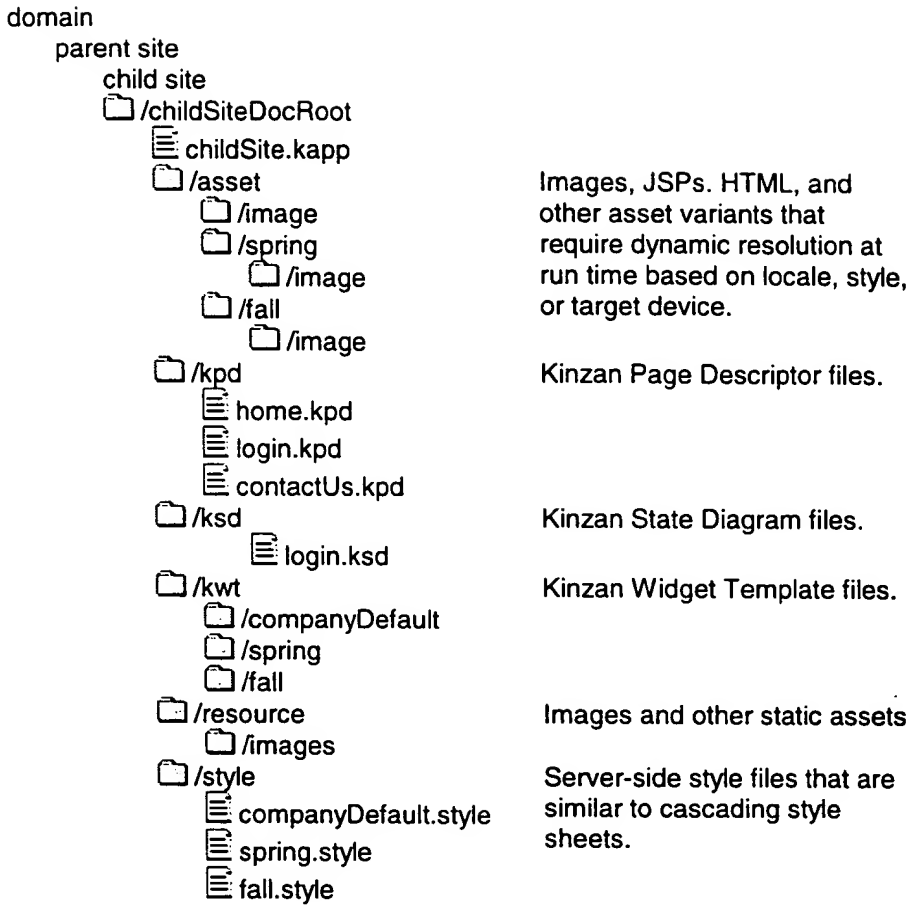
This is not an issue when you are developing an entirely new application. However, it is more common to begin with a generic implementation of the application and customize it for a particular client. In this case, overriding the resource at the application level applies the customization to the entire application at once.

## 5.2 Organizing Files and Folders

The following illustration outlines the directory tree structure convention used on the Kinzan platform.




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN




At compile-time and at runtime, the search path for the site or application includes the search root for the site or application and for each of its parent sites. The search root defines where files for the site are located. The search order is as follows:

1. Look for the file in the document root directory.

2. Look for the file in the *root*/`[subdir]` directory. The type of file being searched for determines which subdirectory is searched. See the tree diagram (above) for specific information about the expected directory structure.

3. Recursively check the parent site's root, applying the same rules

For file types that support locale, style, and/or target device variants, by convention those variants are managed in subdirectories within the appropriate `[subdir]` using the pattern `[locale]/[style]/[target]`. This is a file



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

management convention, and the path to variants (relative to `[subdir]`) must be explicitly called out when referring to them.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 6. The Model Layer

The classic definition of Model/View/Controller architecture describes the model layer as the internal workings of the software. Typically, this layer includes messaging, data storage, integration with legacy systems, and so on.

In the KTP architecture, the model layer consists of the Services Manager and myriad services that are integrated into the Kinzan platform. Examples of integrated services include database systems; EJB, CORBA, and COM components; and external applications such as credit-card processing systems and commercial package-tracking systems.

## 6.1 Overview

When you work within the model layer, you will typically develop and application to integrate into the KTP or develop the glue layer that will link an external application or service into the KTP.

**Skill Level:**

Developing new services and integrating external services into the Kinzan platform requires intermediate to advanced Java programming skills.

## 6.2 Developing Services

When you develop a service, you will either develop an application and integrate it into the KTP or you will develop the glue layer that will link an already developed application or service into the KTP.

Services may be local or remote.

- Local services are collocated—executed in the same memory space as the application that includes them. Local services can also be considered proxyless. Proxyless services do not need to communicate with a back end to perform their functions; all the functionality is contained in the downloaded JAR.
- Remote services are not collocated; invoking their operations requires remote calls. Remote services may be either proxy services or SOAP-enabled services.
    - o Proxy services rely on the functionality residing in the back end to perform their functions. The classes in the downloaded JAR communicate with and invoke operations on a remote server. The StockQuote service is an example of a proxy service.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

- SOAP-enabled services are supported within the native functionality of the platform. They do not require any JARs to be downloaded because the platform automatically generates all classes that are required to communicate with the back end using SOAP. The Siebel and FedEx services are examples of SOAP-enabled services.

To integrate a service into the KTP, you must implement the `KinzanService` interface. This interface declares methods that initialize and close down a service as well as set and get the service name and the service manager for the service.

The `GenericService` abstract base class implements all of the methods declared in the `KinzanService` interface except for `init()` and `fini()`. We recommend that you extend the `GenericService` class when you develop a new service rather than implementing the `KinzanService` interface directly.

Note that accessor and mutator methods within the `GenericService` class are assigned a visibility of `final`. `setName()` and `setServiceManager()` are callback methods that set the name of the service within the framework before the `init()` method is called.

When you extend the `GenericService` class, implement the `init()` method, entering any one-time operations that must be performed before a user can access the service. For remote services, bootstrap the communications protocol in `init()`.

Implement the `fini()` method to close down the service, terminate remote connections, and perform other one-time cleanup operations.

Extend the class to include application functionality, perform calls to an external API, or call those functions in another class. Additional implementation details are dependent on the type of service or application being integrated.

## 6.2.1 Creating a SOAP-Enabled Service

In general, follow the procedures below to create a SOAP-enabled service:

- Define an interface for the service. This interface should extend the `KinzanService` and the `SoapService` interfaces. (Note that `SoapService` is a marker interface, so no implementation is necessary. When the platform resolves a service that implements the `SoapService` interface, it automatically creates a runtime proxy for the service that implements the SOAP-specific calls.)
- If your interface passes or returns Java objects, these objects should implement the `Serializable` interface.
- Implement the class for your interface. The class should extend the `GenericService` class and implement the interface you defined above.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

- Handle all state changes and object communications through the interface. The generic limitations of all RPC mechanisms make this necessary. For example, it would be difficult to SOAP-enable the connection pool service because the connection pool makes direct calls to its connections.

## Example Implementation—FedEx

Given a FedEx tracking number and an account id, retrieve the package destination, date and time of delivery, and the person who signed for it. Here is the interface:

```
public interface FedEXService extends KinzanService, SoapService
{
    public PackageTrackingResults getFedExTrackInfo( String strFedExTrackNo,
                                                      String strFedExAccntId );
}
```

The interface declares the `getFedExTrackInfo` method, which returns the information in a `PackageTrackingResults` object. The `PackageTrackingResults` class serves as a wrapper to the result data. Because an object of type `PackageTrackingResults` is returned to the caller, this class must implement the `Serializable` interface:

```
public class PackageTrackingResults
    implements java.io.Serializable
{
    protected Hashtable iResultSet = null;
    protected Vector iTrackingRecords = null;

    public String getDeliveryToDesc()
    {
        ...
    }

    public String getSignedFor()
    {
        ...
    }

    public String getDeliveryDate()
    {
        ...
    }

    public String getDeliveryTime()
    {
        ...
    }

}
```

The `Serializable` interface is a marker interface used by the `serializer` object to serialize and deserialize the object in a SOAP call. The actual implementation of the service is done in the `LocalFedEXService` class (see the



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

file for details). The important point to note is that the `LocalFedEXService` class must implement the `FedEXService` interface:

```
public class LocalFedEXService extends GenericService
    implements FedEXService
{

   public void init( ServiceDescriptor sd )
       throws Exception
   {
       ...
   }

   public PackageTrackingResults getFedExTrackInfo( String strFedExTrackNo,
                                                     String strFedExAccntId )
   {
       // use FedEx API to track package (no SOAP calls)
       ...
   }

   public void fini()
   {
       ...
   }
}
```

## 6.3 Deploying Services

As they relate to the platform, there are two types of services: static and dynamic.

- Static services are declared in the configuration file for the application, so are known to the application at startup. Static services may or may not be loaded into memory at startup.
- Dynamic services are discovered at runtime. Dynamic services allow you to add functionality without shutting down the server. There are three types of dynamic services:

### 6.3.1 Deploying Static Services

Static services are declared in the XML configuration file for the application. Once the service is declared it becomes available for use at startup. It may be loaded into memory immediately (loadAtStartUp="yes") or when it is needed (loadAtStartUp="no"). The following example declares the logging service and loads it into memory.

```
<service id="LoggingService" loadAtStartUp="yes">
  <class>net.kinzan.logging.LoggingService</class>
  <attribute name="propertiesFile"
                        value="@fileURLPrefix@@wariniroot@/LoggingService.properties" />
```





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T. 760.602.2900 F. 760.602.2910 www.kinzan.com KINZAN

```
</service>
```

## 6.3.2 Deploying Dynamic Services

Deploying dynamic services is a two-step process.

1. Register the service with one or more lookup servers.

2. Deploy the service's files into the core repository.

### Registering the Service

If an application needs to call a service that is not in its configuration file, it consults one or more lookup servers until it locates the appropriate service. Registering a service with a lookup server allows you to add functionality to an application without needing to stop and restart it.

#### Leasing a Service

Once a service is registered with a lookup server, it remains registered until you explicitly remove it from the service or until the server is rebooted. Alternatively, you may lease a service to the lookup server. A leased service is available for a specified period of time, after which it is automatically removed from the lookup server.

> **NOTE:**
> As of this writing, the client site may still be able to use the service proxy after the service has expired. This may change when remote events are fully implemented.

#### Using Multiple Codebases

Using multiple codebases, it is not necessary to package the service proxy and its dependent classes in a single jar. Codebases function in a way that is similar to the CLASSPATH environment variable. Each codebase entry specifies a server and a jar. When the service manager resolves the class dependencies for the service proxy, it looks first at its CLASSPATH; if the dependency is not found, it looks at each codebase in the order specified in the registration file.

#### Using the Registration Tool

The registration tool is a command line utility that accepts an XML file as input. Use this tool to register and remove a service from one or more lookup servers.

Before you can use the registration tool, you must create an XML file to use as input. Following is a sample input file:



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

```
<bootStrap>
    <group name="TestInstance">

        <lookupService>http://chue-2k:8080/soap/rpcrouter</lookupService>

        <service id="StockQuoteService" lease="1186400">
            <codeBase>http://chue-2k:9090/codebase/stockquote.jar</codeBase>
            <codeBase>http://chue-2k:9090/codebase/weblogic51.jar</codeBase>
            <codeBase>http://chue-2k:9090/codebase/weblogicaux.jar</codeBase>

<interface>com.kinzan.example.stockquote.service.StockquoteService</interface>
            <class>com.kinzan.example.stockquote.service.RemoteStockQuoteService</class>
            <attribute name="propertiesFile" value="http://chue-
2k:9090/codebase/StockQuoteService.properties"/>
        </service>

    </group>
</bootStrap>
```

**group** Organizes services into groups. Each KinzanApp is associated with only one group.

**lookupSevice** Specifies the location of the Lookup service. You can specify as many lookup services as you want; the registration tool contacts each specified lookup service and registers each service in the group.

**service** Defines the service. The service id attribute must be unique within the group. The lease attribute is optional. Specify the lease value in seconds.

**codeBase** Specifies the location and name of the service proxy jar or its dependent classes. Codebase entries are not required for SOAP-enabled services.

**interface** Specifies the interfaces implemented by this service. Be sure that each of the specified interfaces is implemented; the tool does not perform any checking.

**class** Specifies the class name that implements the service proxy.

**attribute** Defines one or more attributes for this service.

After creating the input file described above, use a command similar to the following to register the service:

```
java com.kinzan.app.service.registration.Tool -R inputFile.xml
```

## Deploying the Service's Files

### Proxy Service Example

The `stockquote` service is an example of a proxy service implementation. The `stockquote` service uses RMI to talk to its EJB implementation in the backend. Here is how the files should be divided.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

Preferably, deploy `fountainheadExamples jar` on the client side. It should contain the interface for the `stockquote` service and the widget and action files that use the service:

```
com/kinzan/example/stockquote/component/SetDisplayMode.class
com/kinzan/example/stockquote/component/SetEditMode.class
com/kinzan/example/stockquote/component/StockSymbolUpdate.class
com/kinzan/example/stockquote/service/StockQuoteInterface.class
com/kinzan/example/stockquote/service/StockQuoteService.class
com/kinzan/example/stockquote/widget/StockQuoteWidget.class
```

The classes in the component and widget directories implement the action files and widgets. These classes use the `stockquote` service and are independent of the `stockquote` implementation. The `StockQuoteService` and `StockQuoteInteface` files in the service directory define the interfaces to invoke operations and pass data between the `stockquote` service and widgets/action files.

The files in the downloadable `stockquote.jar` need to implement the proxy that communicates with the back end:

```
com/kinzan/example/stockquote/ejb/StockQuoteServiceEJB.class
com/kinzan/example/stockquote/ejb/StockQuoteServiceEJBHome.class
com/kinzan/example/stockquote/service/RemoteStockQuoteService.class
com/kinzan/example/stockquote/service/StockQuote.class
```

The files in the `ejb` directory implement the `ejb` interfaces. The `RemoteStockQuoteService` file implements most of the functionality of the proxy. In addition to this jar, you'll need the `weblogic51.jar` and the `weblogicaux.jar` files.

An easy way to create these two jars is to copy `fountainheadExamples.jar` to the `resin\lib` and `resin\doc` directories, then, make the appropriate changes. Be careful not to remove the properties files from `fountainheadExamples.jar`.

**SOAP-Enable Services Example**

For information on developing SOAP-enabled services, see section 6.2.1.

SOAP-enabled services are supported within the native functionality of the platform. No jars need to be downloaded. On the client side, you need to provide:

- `net.kinzan.external.fedex.FedEXService` for FedEx

On the service provider side you need to install the FedEx and Siebel libraries as well as these classes:

- `net.kinzan.external.fedex.FedEXService`
- `net.kinzan.external.fedex.LocalFedEXService`





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T. 760.602.2900 F. 760.602.2910 www.kinzan.com KINZAN

- `net.kinzan.external.fedex.PackageTrackingResults`
- `net.kinzan.external.fedex.FedEXTransactionMgr`

Note that for demonstration purposes, you don't need to remove these classes from your jar to make them dynamic services.

To make these services dynamic follow these steps:

- Comment out (<!-- -->) the Siebel and the fedex services from `\acme\webroot\web-inf\ini\TestApp.xml`
- Restart your Kinzan instance
- Copy the `siebel.xml` and the `fedex.xml` files to your `\acme\lib directory`.
- Edit the `siebel.xml` and the `fedex.xml` files to change the lookup server URL and the location of the properties file to the appropriate servers.
- Register both services with your lookup server like this: register -R siebel.xml or register -R fedex.xml. Run these commands from the lib directory.
- Copy `Siebel.properties` and `Fedex.properties` to your code base server directory.

## 6.4 Using Existing Services

Once a service has been developed and deployed, it is available for use within the application. Most commonly, you will develop a widget that incorporates the service. For information on developing widgets, see section 7.3.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 7. The View Layer

The classic definition of Model/View/Controller architecture describes the view layer as the visual representation of the state of the model to the user.

In the KTP architecture, the view layer consists of widgets, Kinzan Widget Templates (KWT files), styles, assets, and Kinzan Page Descriptor (KPD) files. The Rendering Service gathers information from all these sources to render and display the page.

## 7.1 The Elements of the View Layer

The elements of the view layer are responsible for rendering a contextually appropriate page and displaying it to the user. The view layer comprises the following elements:

- Widget: An object that represents a visual element. A widget has three parts:
  - Widget Class: a Java bean that contains the attributes for the widget. The Java bean consists of generic data members and set and get methods for those members.
  - Kinzan Widget Template (KWT) file: An XML file that governs the appearance of the widget.
  - KApp file declaration
- Component: This element binds data to the widget. Component elements are declared in the KApp file.
- Kinzan Page Descriptor (KPD) file: An XML file that lists the elements to be included on a page.
- Style file: An XML file that specifies stylistic conventions for the site.
- Asset files: Leaf objects to be included on a page, including images, text, HTML fragments, JSPs, and so on. Assets may have variants that are dynamically resolved based on the current style, locale, or target device.
- Rendering Service: A Java object.

Widgets, templates, pilot components, and data beans are assembled into components. The Rendering Service uses the Kinzan Page Descriptor (KPD) to assemble components into pages.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T. 760.602.2900 F. 760.602.2910 www.kinzan.com KINZAN

## 7.2 Widget Overview

A widget connects the view layer to the model layer, collecting data from external services such as stock quote providers, package tracking systems, and syndicated news services and displaying them on the page.

A widget is a three-part object consisting of:

- A Java Bean
  This container object holds attributes for the widget. It is a Java class that declares one or more private data members and implements set and get methods for those members.
- A Kinzan Widget Template (KWT) file
- A KApp file declaration

Typically, there is an extensive library of widgets available for use. Examples of possible widgets include `Text`, `Image`, `Banner`, `NavBar`, `StockQuote`, `SyndicatedContent`, and `ShoppingBasket`. By defining new presentation templates appropriate to the target device, the same widget can support wireless phones, PDAs, web browsers, and so on.

Widgets can, if necessary, integrate with the Kinzan State Manager, so they can respond to and drive events. This capability simplifies form data routing and processing when there are multiple widgets on a page.

## 7.3 Developing Widgets

Developing a widget typically involves:

- Implementing a set of Java classes that conform to the Widget API to manage the widget
- If necessary, implementing a state diagram and action files to manage the states of a widget that controls application flow
- Creating a KWT file to govern the presentation of the widget
- Declaring the widget in a Kinzan Application (KApp) file

**Skill Level:**

Developing new widgets requires intermediate to advanced Java programming skills.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

## 7.3.1 Components of a Widget

A widget is composed of three Java objects:

- Widget Factory: Object that implements the `WidgetFactory` interface. The Rendering Service calls the widget factory to instantiate new widget instances.
- Widget: Object that implements the `Widget` interface. This object defines the widget properties.
- Widget Proxy: Object that implements the `WidgetProxy` interface. Widget proxies decouple page-specific information, such as style and position, from the widget instance.

### Widgets

The `Widget` interface declares methods to initialize the widget and access the widget name, factory name, properties, and parameters.

As with widget factories, the `GenericWidget` class provides a default implementation of the `Widget` interface and is adequate for developing simple widgets. To develop more complex widgets, extend this class to add widget-specific functionality. Extending the class should only be necessary if the widget requires dynamic information that cannot be managed with widget properties (for example, unique methods to interface with the model layer).

To make a widget available to a site, declare it in the `<widgetList>` section of the KApp file. Widgets declared in the KApp file are available site-wide as shared widget instances. The example below illustrates a shared widget instance declaration in the KApp file. Note that properties defined in the widget list section of the KApp file override like-named properties declared in the widget factory list section.

```
<widgetList>
  .
  .
<widget type="widgetFactoryName" name="sharedWidgetInstanceName">
   <property name="widgetInstanceConfigName1" value="widgetInstanceConfigValue1"/>
   <property name="widgetInstanceConfigName2" value="widgetInstanceConfigValue2"/>
</widget>
  .
  .
</widgetList>
```

Once a shared widget instance is declared in the KApp file, you can reference it in KPDs and KWTs as follows:

```
<widget type="ref" name="sharedWidgetInstanceName" [style="optionalStyle"]/>
```





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

The difference between a widget reference and a widget instance is that the page does not own a local copy of the referenced widget. A referenced widget is instantiated elsewhere; any changes to the widget by the widget owner are reflected in all pages that reference it, unless it is overridden at the page level While the system supports overriding properties and templates, consider instantiating a new widget rather than providing a reference if such behavior is necessary.

For a local widget instance, the widget factory is referenced in the KSPs and KWTs. The syntax for defining a local widget instance is as follows:

```
<widget type="widgetFactoryName">
   <property name="widgetInstanceConfigName1" value="widgetInstanceConfigValue1"/>
   <property name="widgetInstanceConfigName2" value="widgetInstanceConfigValue2"/>
</widget>
```

The Rendering Service instantiates a widget object every time a widget is rendered on a page, so when you extend the `GenericWidget` class, be sure to perform all widget configuration in the `init` method. You can use accessor methods from within the KWT to gain access to dynamic information.

## Setting Widget Properties

In general, you may define whatever properties are appropriate for a specific widget. To define widget properties, enter name/value pairs within `<property>` tags in the widget declaration. Remember that widget factories and widgets are hierarchical, so properties defined in the widget factory declaration are inherited but may be overridden at the shared widget instance and/or local widget instance level. To prevent properties from being overridden, you may assign a visibility attribute of `final` to the widget factory.

Typically, you will define the default presentation for a widget at the factory level and override default presentation values for a shared or local widget instance. Using this technique, it is possible to modify widget presentation at the instance level while benefiting from the reuse of the widget's business and application logic.

While widgets usually have properties that are unique to their implementation, the following properties are reserved for all widgets. The Rendering Service uses these properties at runtime.

**`Template`** (required, set with the `<widgetTemplate>` tag): Defines the variants of the KWT that represent presentation for the widget.

**`JSFILE`** (optional): Contains the full path to a JavaScript file to include in the header portion of the HTML page. Wrap the JavaScript file contents with the `<script>` tag.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

### Widget Templates

All widgets have a template that determines which KWT drives presentation of the widget. This KWT is processed and compiled into a servlet that coordinates with the widget factory and widget instance via a provided `WidgetProxy` class.

As a KWT, presentation is a JSP fragment that is dynamically styled by the runtime environment. You can, therefore, use CSS style classes to drive the server-side styling of the widget presentation.

As a KWT, the widget KWT may also include references to other widgets and to named assets. Like other KWT assets, assets may have variants that are bound to style, device target, locale, and/or structure. The runtime environment dynamically resolves use of the appropriate variant.

Java embedded within the KWT (recall that KWTs are JSP fragments that are processed by the Rendering Service) is used to drive all other presentation logic.

## 7.4 Using Existing Widgets

**Skill Level:**

Using existing widgets requires XML, HTML, and advanced scripting skills.

## 7.5 Developing and Using Style Files

The style file is an XML file that defines stylistic elements to apply to pages that reference the style. The format of the style file closely follows that used in Cascading Style Sheets (CSS), specifying a list of tags and the style attributes for each tag. Multiple classes may be defined in a style, allowing you to define variations of style attributes for given tags.

For example, a company wants to use its corporate colors on the home page for most of the year. However, in the spring and fall of each year it sponsors special events and wants the home page colors to change at these times. Defining separate classes in the `.style` file for Default, Spring, and Fall color schemes allows you to quickly and uniformly change the visual attributes for a page.




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

The Rendering Service applies the style transformation to the flattened JSP file, resulting in a JSP that is ready to be compiled into a servlet for the final application. The style transformation may also include JSP fragments to allow runtime style configuration by the user, for example to allow the user to change the color scheme of the page.

Style transformations occur on the server, so you can benefit from CSS-like capabilities without depending on CSS-enabled browsers. Fonts are supplied by the client installation.

## 7.6 Managing Assets

Assets include objects such as JSPs, text, HTML snippets, and images and may be static or have variants that must be resolved at run time based on locale, style, and/or target device.

Store static assets in a `/resource` directory under the site's document root.

Store variants for assets that must be resolved at run time in an `/assets` directory under the site's document root. Within the `/assets` directory, asset variants should be arranged hierarchically in `[locale]/[style]/[target]` order. See section 5.1 for more information on the Kinzan site hierarchy.

## 7.7 Kinzan Page Descriptors

A Kinzan Page Descriptor (KPD) is an XML file that declares a page's structure, style, and the content for the zones described in the structure being used. A KPD file may also describe a section in another KPD page.

Store top-level KPD pages in the `KPD` directory under the search root.

Store KPDs used to describe sections in the `asset` directory under the search root.

KPD files may also describe a page template (KST files). When KPDs are created as a template, pages that maintain widget references may be created from them at runtime, while creating new widget instances for template elements that are defined as such. In this way, KSTs may be used to load a preformatted page with predefined elements, some of which are controlled at the parent level, others of which are controlled at the site level.

KPD files specify the content to use when assembling a page. This content may include widgets, widget references, and/or container objects (KWTs).





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

## 7.8 Th Rendering Service

The Rendering Service is responsible for dynamically assembling KSPs when requests are sent to the web server.

The Rendering Service is fully dynamic, and may thus (with proper tools) be manipulated and configured on the fly, without having to recompile site pages or site modules. This rapid configurability is a key feature of the KSP Framework.

## 7.9 Rendering Service

The Rendering Service internally uses the Model-View-Controller (MVC) paradigm to render pages.

The Model is the set of Java classes supplied with the widgets. These Java classes are responsible for creating the widgets, retrieving any data required by them and implementing any business logic. Note that internal to the rendering environment, all components (pages, widgets, containers, etc.) are modeled as widgets.

The View is the visual representation of the widget. In the rendering system, this is embodied in the corresponding widget template KWT file, which is processed into a JSP file for the widget. When the processed KWT file is compiled into a servlet and executed, the servlet interrogates the widget and uses the data contained in the widget to create a visual representation of it.

The Controller is the rendering servlet. The rendering servlet processes requests by matching a request to a model and view.

The figure above presents a detail depiction of how the widget Java classes and servlets are used by the rendering system to process a request. A brief description of each component follows:

**Rendering Servlet**: The rendering servlet processes requests to render pages. It forwards the request to the rendering service and invokes the appropriate servlet for the page.

**Rendering Service**: The rendering service is responsible for building the model of a page. It uses the Runtime Persistence service to retrieve data from the Runtime datastore. For each component on the page, the rendering service uses this data to invoke the appropriate widget factory and create the corresponding widget. The page model is returned as a tree structure where the root of the tree is the page widget.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

**Runtim P rsistence Servic** : The Runtime Persistence service encapsulates all operations for the Runtime datastore as a service. The datastore is only accessible through the Runtime Persistence Service.

**Widget Factory**: The widget factory is an implementation of the factory pattern. The Rendering Service uses the widget factory to instantiate a widget without knowing which type of widget is being created or what steps are required to create the widget. The widget factory is managed by the Rendering Service. The Rendering Service creates and initializes the widget factory once during the first request for a particular widget. The Rendering Service uses the same widget factory for any subsequent requests for the same widget type.

**Widget:** The widget object contains all the data required to render the widget. At runtime, a widget also provides access to all its defined properties.

**Widget Proxy**: The widget proxy is used by the Rendering service as a placeholder for the widget. The Rendering service uses information in the widget proxy to style and place the widget on the page. The Widget Proxy relieves the widget designer from maintaining page-specific information.

**Widget Template KWT**: The widget template KWT (once processed and compiled into a servlet) translates the data stored in a widget instance into its visual representation.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 8. The Controller Layer

The classic definition of Model/View/Controller architecture describes the controller layer as the means by which the user provides input or changes the state of the model.

In the KTP architecture, the controller layer consists of the State Manager, Kinzan State Diagrams (KSD files), and Action files.

## 8.1 The Elements of the Controller Layer

The elements of the controller layer are responsible for accepting user input and tracking the user's progression through the application embedded in the Web site. The controller layer comprises:

- Kinzan State Diagram (KSD) files: XML files that define discrete steps in an application flow with decision points that trigger transitions to different sections of the application.
- Action files: Java objects that implement application logic. Action files connect the controller layer to the model layer.
- Request context: Java objects that track the user's progress through the application. The request context object provides input to the decision points of the KSD.
- State Servlet: Java objects that collect user input and pass the information to the State Manager for processing. When the call to the State Manager returns, the State servlet dispatches a request to the appropriate Kinzan Page Descriptor file in the view layer, or redirects to a new URL, depending on what was returned by the State Manager.
- State Manager: A Java object that controls the execution of a domain.

## 8.2 Developing and Using State Diagrams

Kinzan State Diagrams (KSD files) are XML files that define the different states in an application and how events drive application functionality. They are used to assemble, organize, and connect the various modules that make up the controller tier.

State diagrams illustrate action states and display states.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

- Action states use modular action files to execute elements of application logic; the result of one operation drives the next operation. For instance, an `xApproveCreditCard` action state may invoke an `xShipProduct` action state if a transaction is approved, or invoke an `xRejectOrder` action state if the credit card is declined.
- Display states show the user the next step in the application. Once all backend processing is done, the state diagram specifies the appropriate KPD to invoke. In the previous example, the user sees an `OrderConfirmation` display state if the card is approved and the product shipped and a `ReenterCreditCard` display state if the card is rejected.

Overall application flow is controlled via the state diagram file, with action files and page descriptors available to be shared within and between applications to encourage reuse.

The state diagrams that an application requires are defined in the KApp file as follows:

```
1  <stateManager>
2    <defaultState wizard="defaultWizardName" state="defaultStartState"/>
3
4    <wizard name="wizardName">
5      <file>wizardName.wizard</file>
6    </wizard>
7
8  </stateManager>
```

Line 2 — Set the default starting wizard and start state for the application

Lines 3–5 — Define the wizard name and file to be made available to the application

KSDs defined in the KApp file are loaded into the runtime environment and can be invoked by KPDs. Like other modules, state diagrams are also inherited from parent sites to child sites, encouraging reuse.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T. 760.602.2900 F. 760.602.2910 www.kinzan.com KINZAN

# 9. The KApp File

The Kinzan Application file (KApp file) acts as the makefile for a site. This XML file provides site information and describes the objects that are available to pages within the site. The following sections introduce the various sections of a KApp file.

## 9.1.1 Site Information

Each KApp file corresponds to one site, which in turn is assigned to a particular level in the runtime site/application space. The XML sample below illustrates how to supply site information in the KApp file.

```
1     <site name = "sitename" visibility="visibilityLevel">
2        <parent>parentSitename</parent>
3        <domainList>
4           <domain name="siteDomain.com">
5              <documentRoot>siteDocumentRoot</documentRoot>
6           </domain>
7        </domainList>
8        <parent>parentSiteName</parent>
9     </site>
```

Line 1 — Set the site's name, which must be unique. Visibility establishes whether the site and its content are visible to child sites. Acceptable values are public, private, and protected.

Lines 3–7 — Set the domain and the document root directory for the site. These values are used at runtime to access this site (i.e., `http://domain.com/documentRoot/…`). A site may have many domains in the domain list.

Line 8 — (Optional) Link the site to the parent site by name.

At compile-time and at runtime, the search path for the site or application includes the search root for the site/application each of its parent sites. The search root defines where files for the site are located. The search order is as follows:

3 - Look for the file in the document root directory

4 - Look for the file in the `root/{subdirs}` directory (see below)

5 - Recursively check the parent site's root, applying same rules

The type of file being searched for determines which subdirectory is searched.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

| [subdir] | File Type |
|---|---|
| /style | Styles |
| /ksp | KSPs |
| /kwt | KWTs |
| /wizard | Wizards |
| /asset | Other assets (images, JSPs, HTML) |
| /resource | Miscellaneous resources that do not require dynamically resolved variations based on style, target device, or locale. |

For file types that support locale, style, and/or target device variants, by convention those variants are managed in subdirectories within the appropriate [subdir] using the pattern: [locale]/[style]/[target]. This is a file management convention, and the path to variants (relative to [subdir]) must be explicitly called out when referring to them.

## 9.1.2 Structure and Style Mappings

The KApp file determines which structures and styles are available to be used for the site. Structure and style names are mapped to their associated files and described as follows:

```
1     <structureList>
        .
        .
 2      <structure name="structureName">
 3         <variant target="HTML">structureFile.structure</variant>
 4         <description>This is a structure</description>
 5      </structure>
        .
        .
 6    </structureList>
 7    <styleList>
        .
 8      <style name="styleName">
 9         <styleFile>styleFile.style</styleFile>
10         <description>This is a style.</description>
11      </style>
        .
        .
12    </styleList>
```

Lines 1–6 List the structures to make available to this site. Pages in the site may use the structures defined at this level and structures defined at any parent level.

Lines 2–5 Define a structure. The name must be unique within this site; the description is optional. The file is searched for as specified in the previous section. A structure




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

may have several variants based on a combination of target device type, style, and locale. The appropriate variant is determined dynamically when pages are rendered.

Lines 7–12 List the styles to make available to this site. Pages in the site may use the styles defined at this level and styles defined at any parent level.

Lines 8–11 Define a style. The name must be unique within this site; the description is optional. The file is searched for as specified above.

When the structure file is loaded into the runtime environment, it is parsed into a container JSP file that reflects the same structure. In the runtime environment, the structure is actually represented as a widget instance that is sharable across this site and any descendent sites. This widget drives the recursive assembly of a page.

Styles are parsed and loaded into the runtime environment and are later applied by the Rendering Service to stylize all processed elements (KWTs, widgets, structures, etc.).

Structures and styles declared in the KApp file are loaded into the runtime environment and made available to this site and descendent sites. The KSP file for that page defines the style and structure to be used for a given page.

Structure and style files are discussed in detail below.

## 9.1.3 Asset Mapping

Assets are named resources that are available for use within the site. Assets may have multiple variants based on the active style, locale, and/or target device type for which a page is being rendered. Assets are mapped to the appropriate file in the KApp file as follows:

```
1     <assetList>
        .
        .
2       <asset name="imageAssetname" visibility="visibilityLevel">
3          <variant target="HTML">image/assetfile.gif</variant>
4          <variant target="HTML" style="styleName">
5             image/styleName/assetfile.gif
6          </variant>
7          <variant target="WML">image/wml/assetfile.gif</variant>
8          <description>An asset</description>
9       </asset>
        .
        .
10    </assetList>
```

Line 2 Specify the asset name, which must be unique within the site.




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

Lines 3–7 Declare the variants for the asset and the file they are bound to. Variants may be based on any combination of locale, style, and target device type.

Line 8 Describe the asset. (Optional)

Assets may have variations depending on style, locale, and target device. Style is always preferred when finding the best match for an asset. A default (non-style/locale/target-linked) variation is not required, but is recommended.

Grouping assets is a powerful technique for supporting localization, branding, and personalization. You can use generic assets when you develop pages and applications and still deliver a customized presentation based on the person viewing a site.

## 9.1.4 Widget Mapping

Widgets are made available to a site by specifying information for their factory in the KApp file. In addition, widget instances declared in the KApp file are available to all pages in a site as a site widget. Site widgets may be referenced by any page in this site or children site, and be centrally managed by this site.

See section 1 for information on building and using widgets.

To make widget factories available to a site, define them in the KApp file as follows:

```
1    <widgetFactoryList>
       .
       .
2      <widgetFactory name="widgetFactoryName">
3         <class>com.domain.WidgetFactoryClassName</class>
4         <widgetTemplate>
5            <variant target="HTML">widgetTemplate.KWT</variant>
6         </widgetTemplate>
7         <property name="propertyName1" value="propertyValue1"/>
8         <property name="propertyName2" value="propertyValue2"/>
9         <widgetIncludeList>
            .
            .
10            <widget type="incWidgetFactoryName" binding="bindingName">
11               <property name="incPropertyName1" value="incPropertyValue1"/>
12               <property name="incPropertyName2" value="incPropertyValue2"/>
13            </widget>
                .
                .
14         </widgetIncludeList>
15      </widgetFactory>
       .
       .
16   </widgetFactoryList>
```

Line 2 Specify a widget factory. The name must be unique within this site.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T 760.602.2900 F 760.602.2910 www.kinzan.com KINZAN

Line 3 Specify the class that defines the interface for this widget.

Lines 4–6 Specify the default template to be used for the widget presentation. Widget templates may have many variants based on target device type, style, and/or locale, and may be overridden by widget instances.

Lines 7–8 Set default properties for the widget. A list of these parameters may be provided if needed. The `<property>` tag defines configuration values for the widget factory. Configuration values are widget dependent and may be overridden by widget instances.

Lines 9–14 (Optional) A widget may contain one or more other widgets, as defined in the `<widgetIncludeList>`. Included widgets are defined the same as other widgets, with the addition of a `bindingName` which is used by the parent widget to coordinate and communicate with the included widget. If included widgets are defined with the widget factory, they define the default include widgets for widget instances, and may be overridden at the instance level.

Shared site widget instances may also be defined in the KApp file. Uses for this may include defining a site-wide logo or copyright:

```
1     <widgetList>
        .
        .
2       <widget type="widgetFactoryName" name="widgetInstanceName">
3          <property name="instancePropertyName" value="instancePropertyValue"/>
4       </widget>
        .
        .
5     </widgetList>
```

Line 2 Specify the widget instance name and provide the factory name on which the widget is based.

Line 3 Specify properties for the widget instance. For example, for an image widget this may be the image asset to display.

The syntax for defining a widget instance is the same across all files (KSP, KWT, etc.). The widget factory is referenced and widget template and properties are set. If you are defining a site-wide, shared widget instance in the KApp file for the site, name the instance . Other objects can then reference this shared widget instance via the `ref` widget type using `widgetInstanceName`.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T· 760.602.2900 F· 760.602.2910 www.kinzan.com KINZAN

# 10. Security

## 10.1 Component Security

### 10.1.1 Component Level Security (Level 1)

An application builder will be able to restrict a list of component instances within a site or page. The application builder will also define the Application Security Roles that will be granted permission to access the restricted components. When a page is rendered for a particular user, the restricted components will be rendered with a "No Authority" message (configurable).

### 10.1.2 Component Mode Security (Level 2)

A component builder can set up permissions on the modes of the component at time of assemble. The application builder will be responsible for mapping application security roles to component permissions.

### 10.1.3 Component Model Security (Level 3)

The business rule bean that creates a model will have an optional Security interface that it can implement to provide security information on what roles have read, or read/write on each attribute. Action beans that are mapped to the model attributes will reflect the security on the Model itself that the JSP component form library will use in rendering forms specific to user security privileges.

## 10.2 ACL Overview

### 10.2.1 What is an ACL?

An ACL is a list of access limits to be enforced by the security system. If a secure application state is akin to a locked door, the ACL is the list of all the doors with locks on them. A logged-in user must have the right permission to pass through a locked door. A generic ACL is a list of access controls for a set of resources. In the KTP framework, the states of the application are the resources being protected.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

### 10.2.2 Advantag of ACLs

ACLs allow you to control user access to the functionality of the application. The most basic use of ACLs is to limit access to a group of insiders; those users that are members of a group (e.g., a company) can be allowed functionality that guests are disallowed. A more complex approach encapsulates all key roles of the business process and maps them to functionality appropriate to their business function.

### 10.2.3 Applying ACLs to Wizard Files

The most basic use of ACL is to protect an entire wizard within a full KTP state manager application.

```
<?xml version="1.0"?>
<!DOCTYPE ktp:wizard SYSTEM "http://www.kinzan.net/dtd/wizard.dtd">
<!-- A very basic ACL Example using Security Infrastructure -->
<ktp:wizard name="Secret" startState="Secret" xmlns:ktp="http://www.kinzan.net">
    <ktp:access role="test" />
    <ktp:displayState name="Secret" ksp="Secret" />

</ktp:wizard>
```

A more specific form of ACL setting is to protect individual states within a wizard.

```
<?xml version="1.0"?>
<!DOCTYPE ktp:wizard SYSTEM "http://www.kinzan.net/dtd/wizard.dtd">
<!-- A very basic ACL Example using Security Infrastructure -->
<ktp:wizard name="SecurityExample" startState="Home" xmlns:ktp="http://www.kinzan.net">
    <ktp:displayState name="Home" ksp="Home">
            <ktp:transition event="EDIT_USER_PROFILE" state="EditUserProfile"/>
    </ktp:displayState>

    <ktp:displayState name="EditUserProfile" ksp="EditUserProfile">
            <ktp:access role="SecurityExampleUser" />
            <ktp:transition event="DONE" state="Home"/>
    </ktp:displayState>
</ktp:wizard>
```

In the preceding sample wizard file, the home page state is unguarded to welcome guests, but the function of editing a user profile, for example to change the home address, requires that the user be logged in and assume the `SecurityExampleUser` role.

## 10.3 Secure Tokens

Furthermore, secure tokens may be used.





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 11. Logging Service

The `LoggingService` integrates the IBM JLog package into the KTP environment and implements two loggers: a trace logger and a message logger.

- The trace logger is intended to be used by KTP developers as a debugging tool.
- The message logger is intended to send messages to users of Kinzan-developed systems, such as system administration personnel and web page designers, and is capable of being localized.

The trace logger and the message logger can each be run at the system level and at the application level.

## 11.1 System Loggers

System loggers are visible across multiple KinzanApp instances. To access a system logger, use the `GetSystemTraceLogger` and `GetSystemMessageLogger` static methods.

`net/kinzan/logging/LoggingService.properties` stores system logger properties. (On the Kinzan network, there's an example of this file in the resource directory of the StarTeam acme project.) If the system logger cannot locate this file, it defaults to full synchronized logging and sends output to the console. However, if the file exists, the output destination must be specified explicitly.

## 11.2 Application Loggers

Application loggers are created by a specific instance of the LoggingService, therefore they are only visible within a KinzanApp. They can be accessed in the usual way:

```
LoggingService logService = KinzanApp.GetService( "LoggingServiceName" );
Logger traceLogger = logService.getTraceLogger();
```

Use the application loggers whenever possible. They provide greater flexibility in that you can have different settings for each application logger.

## 11.3 Logging Output

You can configure each logger in the service to send its output to one or more supported devices. Currently supported devices include the console, files, and




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

sockets. For each output device, you can configure its filter and message formatter.

## 11.4 Setting Properties

The logging service supports the following properties:

**`Logger.Description`** (optional)
A brief description that identifies the logger.

**`Logger.Organization, Logger.Product, Logger.Component, Logger.Server, Logger.Client`** (optional)
Used by the formatters to assemble a line in the logger.

**`Logger.Message.MessageFile`**
Sets the path to the resource bundle the message logger should use.

**`Logger.Trace.Logging, Logger.Message.Logging`**
When the appropriate property is set to `true`, turns on trace or message logging.

**`Logger.Trace.Synchronous, Logger.Message.Synchronous`** (optional)
When set to `true`, the call to the Logger object becomes a blocking call. During development, you can set these to `true`, but in production, these values should be set to `false`. The default is `false`.

**`Logger.Trace.Console, Logger.Message.Console`**
When set to `true`, directs logger output to the console. The default is `false`.

**`Logger.Trace.Console.Formatter, Logger.Message.Console.Formatter`** (optional)
If `Logger.Trace.Console` or `Logger.Message.Console` are set to `true`, these fields specify the class used to format a message. Valid options are:

- `com.ibm.logging.EnhancedFormatter`
- `com.ibm.logging.EnhancedTraceFormatter`
- `com.ibm.logging.TraceFormatter`
- `net.kinzan.logging.SingleLineTraceFormatter`

**`Logger.Trace.Console.Mask, Logger.Message.Console.Mask`**
Use these fields to set the message types that are allowed through the filter for the Console device. For instance, if you specified: `TYPE_EXIT TYPE_LEVEL1 TYPE_LEVEL3` for the Trace logger, only messages from the Trace logger with type `EXIT`, `LEVEL1`, and `LEVEL3` are displayed on the console. For valid type values, see the `IRecordType` interface.

**`Logger.Trace.File, Logger.Message.File`** (optional)
When set to `true`, directs logger output to a file.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

**Logger.Trace.File.FileName, Logger.Message.File.FileName**
If `Logger.Trace.File` or `Logger.Message.File` are set to `true`, these fields specify the filename to which the logger writes messages. Do not use backslashes ( \ ) in your path, the JLog package does not like them.

**Logger.Trace.File.MaxFileSize, Logger.Message.File.MaxFileSize**
If `Logger.Trace.File` or `Logger.Message.File` are set to `true`, these fields specify the maximum file size in KB. Once the file reaches this size, the logger creates a new one. The default is 1MB.

When the system creates the first new file, it appends 1 to the original filename; it appends n+1 to each succeeding file (so `myfile.log` becomes `myfile1.log`, `myfile2.log`, etc.).

**Logger.Trace.File.MaxNoFiles, Logger.Message.File.MaxNoFiles**
If `Logger.Trace.File` or `Logger.Message.File` are set to `true`, these fields specify the maximum number of files that the logger should keep when it rotates the log files.

**Logger.Trace.File.Formatter, Logger.Message.File.Formatter**
See `Logger.Trace.Console.Formatter`

**Logger.Trace.File.Mask,Logger.Message.File.Mask**
See `Logger.Trace.Console.Mask`

**Logger.Trace.Socket, Logger.Message.Socket**
When set to `true`, the system directs log output to another machine. See the JLog documentation for instructions on setting up the server daemon that should listen to these messages.

**Logger.Trace.Socket.ServerName,Logger.Message.Socket.ServerName**
When `Logger.Trace.Socket` or `Logger.Message.Socket` are set to `true`, these fields specify the server that messages are routed to.

**Logger.Trace.Socket.Port,Logger.Message.Socket.Port**
When `Logger.Trace.Socket` or `Logger.Message.Socket` are set to `true`, these fields specify the port numbers that the logging daemon is listening to.

**Logger.Trace.Socket.Formatter,Logger.Message.Socket.Formatter**
Please see `Logger.Trace.Console.Formatter`

**Logger.Trace.Socket.Mask,Logger.Message.Socket.Mask**
Please see `Logger.Trace.Console.Mask`

## 11.5 Using the Logging Service

`StateManagerServlet`, `BasicLogComponent`, `RequestContext` and `StateEngine` can now use the LoggingService.




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

Here are some conventions to use:

- Loggers should not be static because they are linked to an application instance.
- If your loggers are member variables, make them **protected** rather than **private** so child classes do not need to construct their own loggers.

Define the logging service as the first service in your application properties file, like this:

```
New Logging service setup:

        <service id="LoggingService" loadAtStartUp="yes">
            <class>net.kinzan.logging.LoggingService</class>
            <attribute name="propertiesFile" value="file:/E:/projects/Acme/webroot/web-
inf/ini/LoggingService.properties"/>
        </service>
```

The following example outlines the steps necessary to include logging in your application. It illustrates a simple Java class with a `main()` method and the code that is necessary to make LoggingService available to any class in use by the main class.

In `main()`, include the following line of code:

```
    KinzanApp.InitIntance( "myInstanceName", myPropFileName );
```

Argument 1 A unique string that identifies an instance of the KinzanApp.

Argument 2 (String) The name of the properties file for the Instance. `TestApp.properties` is a sample of this file.

Somewhere in your classes, you can get a service such as a the LoggingService like this:

```
LoggingService loggingService = (LoggingService) KinzanApp.getService( "LoggingService" );
```

Argument 1 The string that you pass in is the same string you specified when you declared the service in the application properties file (see above).

If you don't want to use the `KinzanApp` class, you can get a logger by using the LoggingService directly, like this:

```
    Logger myLogger = LoggingService.LoadLogger( loggerName, loggerPropertyFileName );
```





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

Argument1 The logger name. This name is used to look for the properties in the logger properties file. For example, if you use the `myLogger` as the logger name, `Logger.myLogger.Console`, `Logger.myLogger.Logging`, etc. should appear somewhere in the properties file.

Argument 2 The name of the logger properties file.

If you take this approach, you have to devise your own mechanism to get access to the logger you created from a class.

The Logging service returns Logger objects. This is the Logger object defined in the IBM JLog package.

Here is an example on how to use the class:

```
// get loggers
LoggingService loggingService = requestContext.getService( "MyFavoriteLogger" );
Logger traceLogger = loggingService.getTraceLogger();
Logger messageLogger = loggingService.getMessageLogger();
traceLogger.text( IRecordType.TYPE_LEVEL1, "MyClassName", "MyMethodName", "Hi MOM!" );
```

- The standard name for the logging service is `LoggingService`.
- Constant classes, such as `IRecordType`, are not implemented.
- By convention, logger names are `iTraceLogger` and `iMessageLogger`.

Be sure to use the correct record type when you log entry and exit messages. This makes it easy to turn them on and off later. For exit logs, use `IRecordType.TYPE_EXIT` and for entry use `IRecordType.TYPE_ENTRY`.

## 11.5.1 Filters

`AttributeFilter` allows you filter any attribute on a `LogRecord`, either through inclusion or exclusion.

The following JLog attributes are standard:

- `loggingClass`
- `loggingMethod`
- `organization`
- `product`
- `component`
- `client`
- `server`



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

The first two, `loggingClass` and `loggingMethod`, are typically the values you pass in to `logger.entry`, `logger.exit`, `logger.text`, `logger.exception`, etc. (The other five are typically configured on a per-LoggingService basis. They will become more important as we start deploying multiple web apps.)

Using `loggingClass` and `loggingMethod`, you can include only specific classes and/or methods you want to see and filter out all others or the opposite, exclude specific classes and/or methods.

The filter setting is in the default properties for both `GlobalSystem` and `LoggingService`. They've been commented out, so the typical behavior is still in play. But if you uncomment the line below for desired `LoggerName` and `LoggerType`, you'll activate the `attributeFilter`.

```
#Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute=loggingClass loggingMethod
organization product component client server
```

You can either exclude the listed attribute values or include them:

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.match=include
```

or

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.match=exclude
```

You can also specify an Any or All filter by setting the `matchAll` value - match one value of each attribute (either all attributes or just one). To match all attributes listed, use `true` or to match any attribute listed use `false`.

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.matchAll=true
```

For each attribute you can specify a space-separated list of values as an include or exclude list that must be matched by the logger.

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.loggingClass=
                                    net.kinzan.security.auth.module.JndiLoginModule
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.logginMethod=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.organization=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.product=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.component=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.client=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.server=
```





2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

## 11.5.2 Additional Options

You can define more than one logging service if you need to. The biggest disadvantage with this approach is that not all classes have access to a request context. In these situations, the code to get the Logging service becomes convoluted:

```
AppContext appContext = KinzanApp.GetAppContext( "MyAppName", null );
LoggingService loggingService = KinzanApp.GetService( appContext, "MyFavoriteLogger" );
```

This implies that you have access to the Application instance name and the Logging service name.



2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# 12. Further Considerations

### 12.1.1 Widgets

```
<widget type="widgetFactoryName" [style="optionalStyle"]>
   <widgetTemplate>
      <variant>...</variant>
   </widgetTemplate>
   <property .../>
   <widgetIncludeList>
      <widget binding="widgetBindingName">
         .
         .
         .
      </widget>
   </widgetIncludeList>
</widget>
```

This tag defines a widget instance directly. You may supply widget-dependant properties and templates when you instantiate a widget, otherwise the properties are inherited from the widget factory. If you specify an optional style, the widget instance representation will use it.

If the page designer instantiates a widget in a page, the page owns a local copy of the widget.

Supplying a name for the widget instance is neither required nor recommended. The compiler automatically assigns a unique name to ensure that there are no collisions in the runtime environment.

Additionally, you may specify a `widgetIncludeList` for the widget instance if necessary.

### 12.1.2 Widget References

Shared widgets that are instantiated and named in the KApp file may be referenced by name using a widget of type `ref`.

```
<widget type="ref" name="siteWidgetInstanceName" [style="optionalStyle"]/>
```

### 12.1.3 Container objects

An example of a container object is a KWT:




2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

```
<widget type="kwt">
   <widgetTemplate>
      <variant target="HTML">kwtFile.kwt</variant>
   </widgetTemplate>
</widget>
```

The KWT widget provides the page designer with an easy way to include a template file that is not necessarily bound to a widget type. A KWT file allows someone to quickly design a section containing HTML, JSP, and other widgets or widget references, including other KWTs. KWT files are styled at compile time to use the style of the object they are contained in. They are searched for in the KWT directory under the search root at compile-time. Appendix A presents a sample KWT file and its syntax.

A section KPD, on the other hand, is a more structured container that takes advantage of reusable components such as style and structure and may contain other KPDs, KWTs, and leaf widgets.

If your application requires flexibility and the section's content needs to be dynamic, we recommend including a KPD as a section. Otherwise, a simple KWT is acceptable, especially if no specific style is required and most of the content is static anyway.

Following is an example of referencing a section KPD:

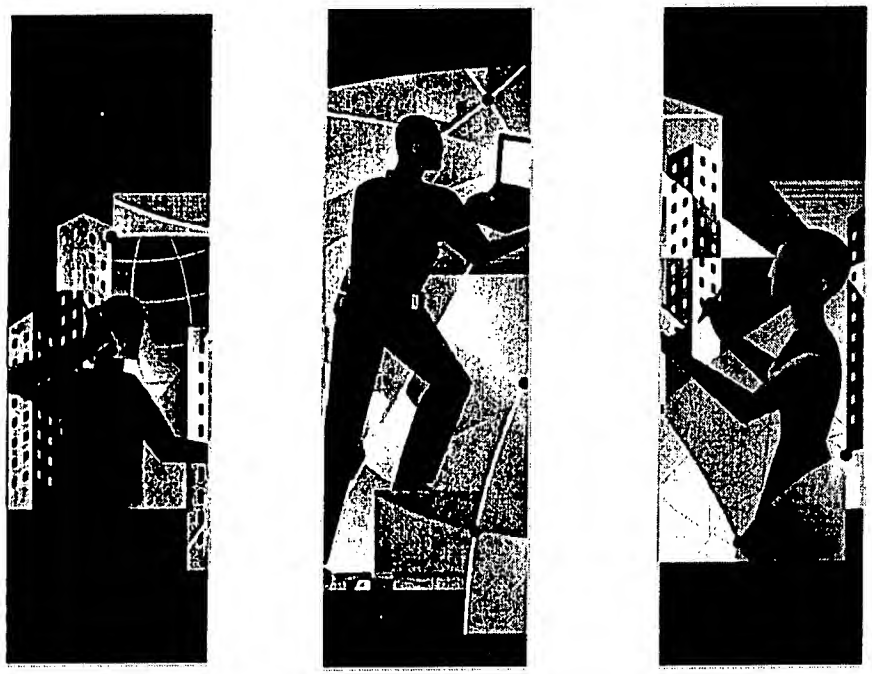
```
<widget type="KPD">
   <property name="KPD" value="nameOfKPD.kpd"/>
</widget>
```





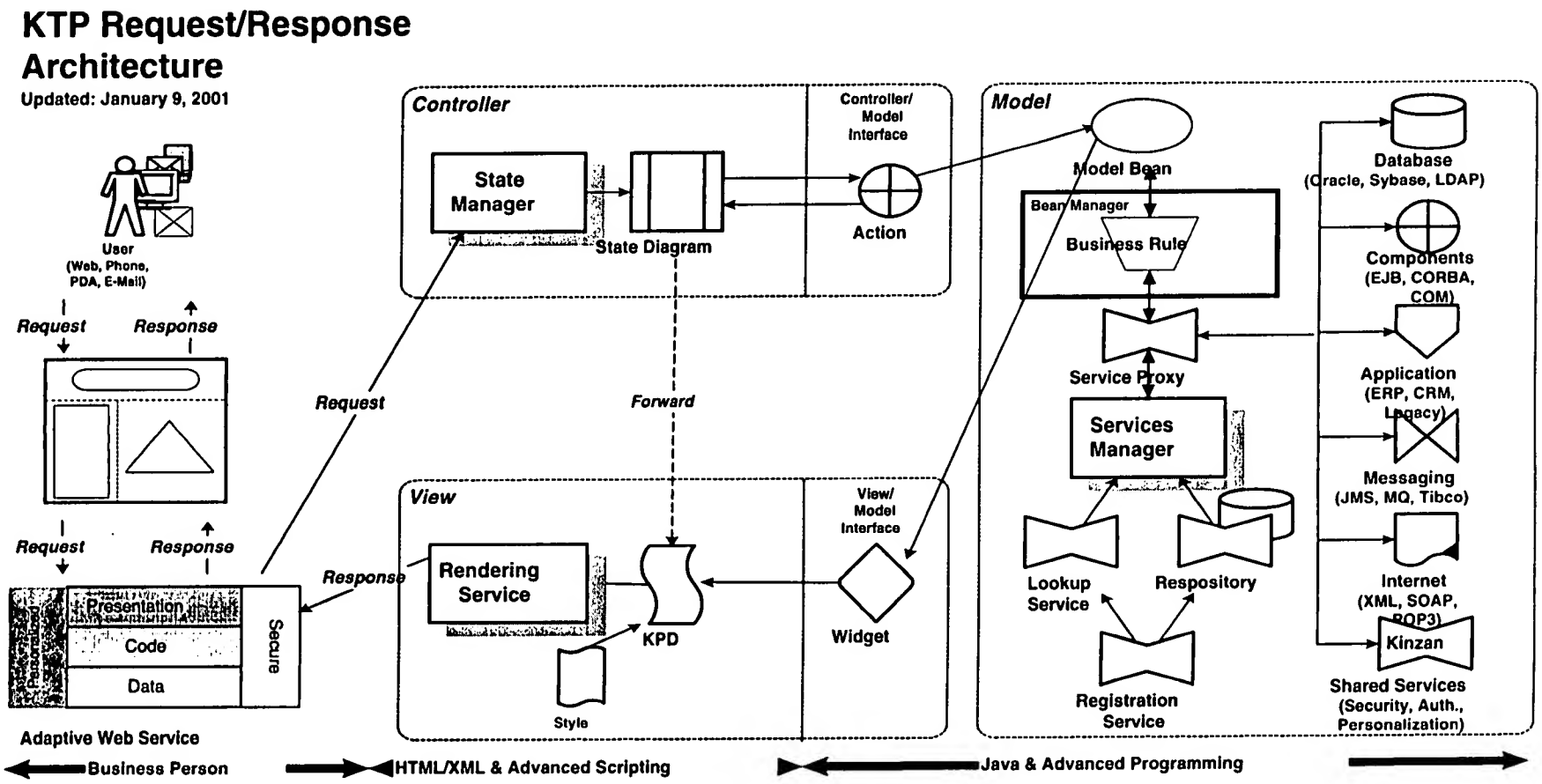
2111 Palomar Airport Road Suite 250 Carlsbad CA 92009 T: 760.602.2900 F: 760.602.2910 www.kinzan.com KINZAN

# Component Architecture

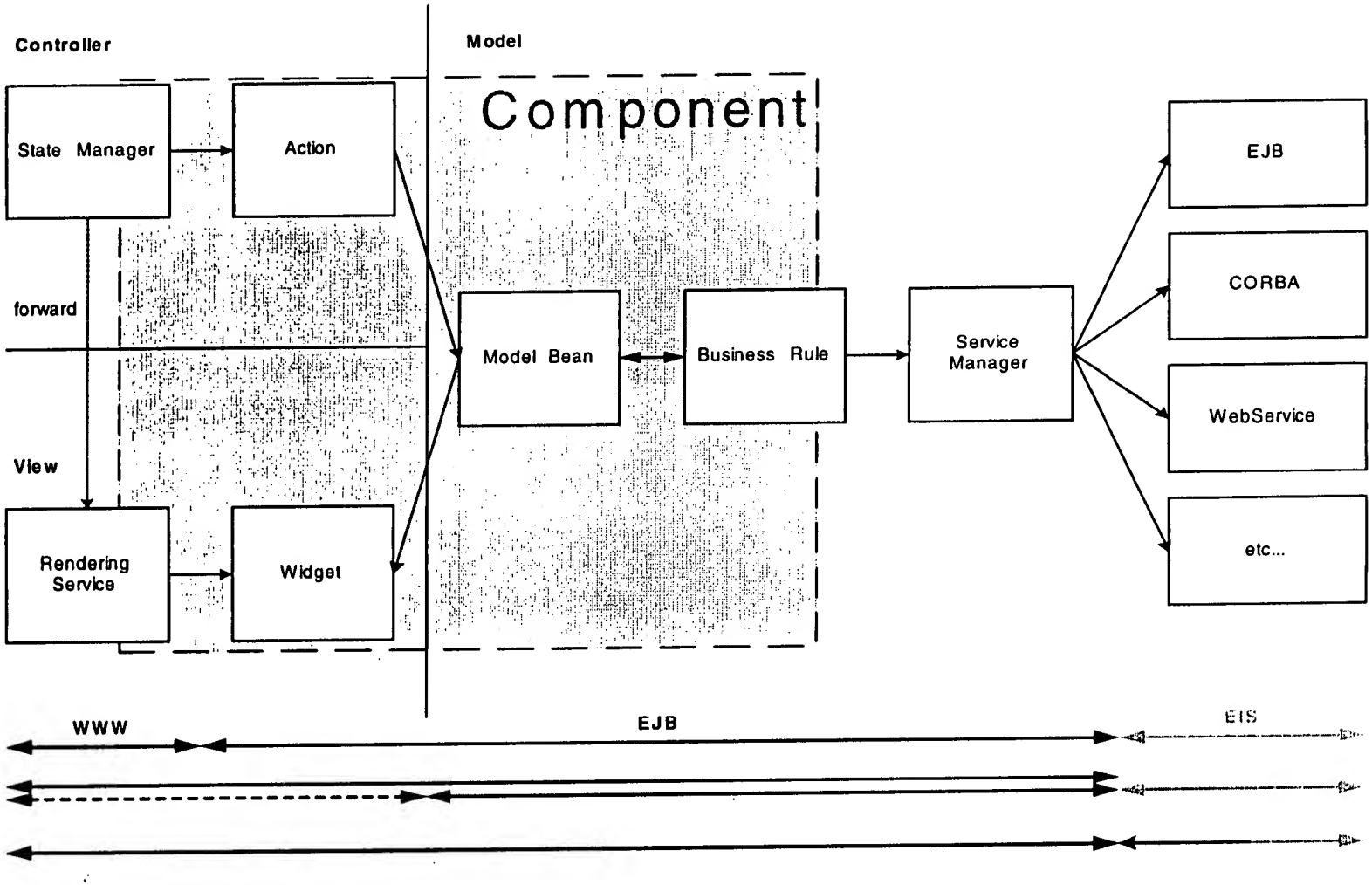


January 11, 2001

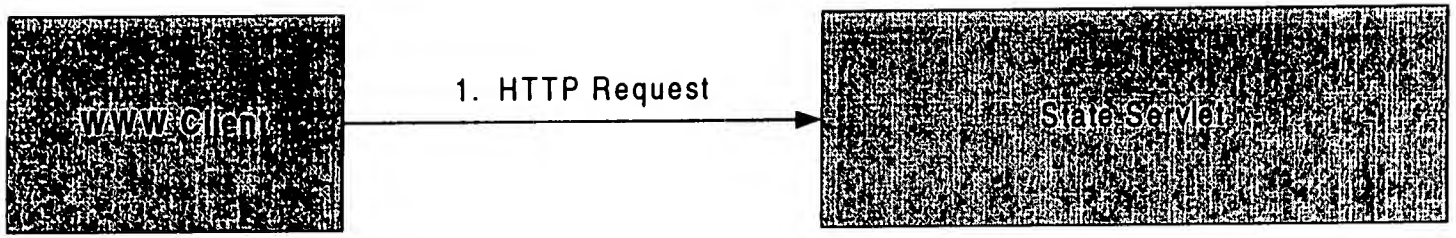
# KTP Request-Response Architecture



# KTP Component Overview



# The Request




The HTTP Request is in the form:

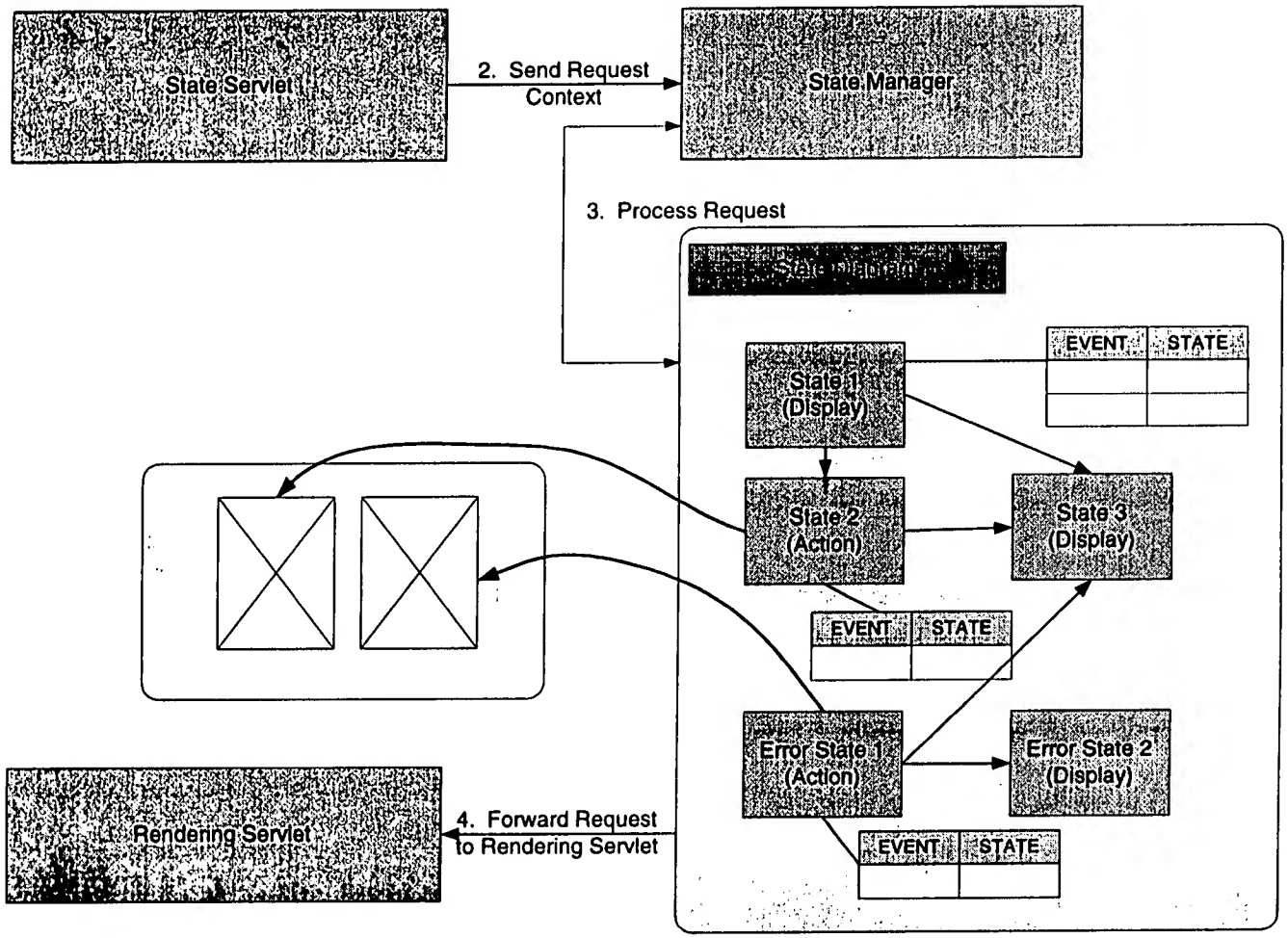
http(s)://address:port/stateServletName/application/ksd/state?requestContextID=RQID&EVENT=event

where

stateServletName = alias for state servlet in the servlet runner
application = name of the application/site that the state diagram exists in*
ksd = name of the state diagram to run*
state = state in the state diagram to run*
RQID = Identifier for the current request context/session*
event = The name of the event to post.

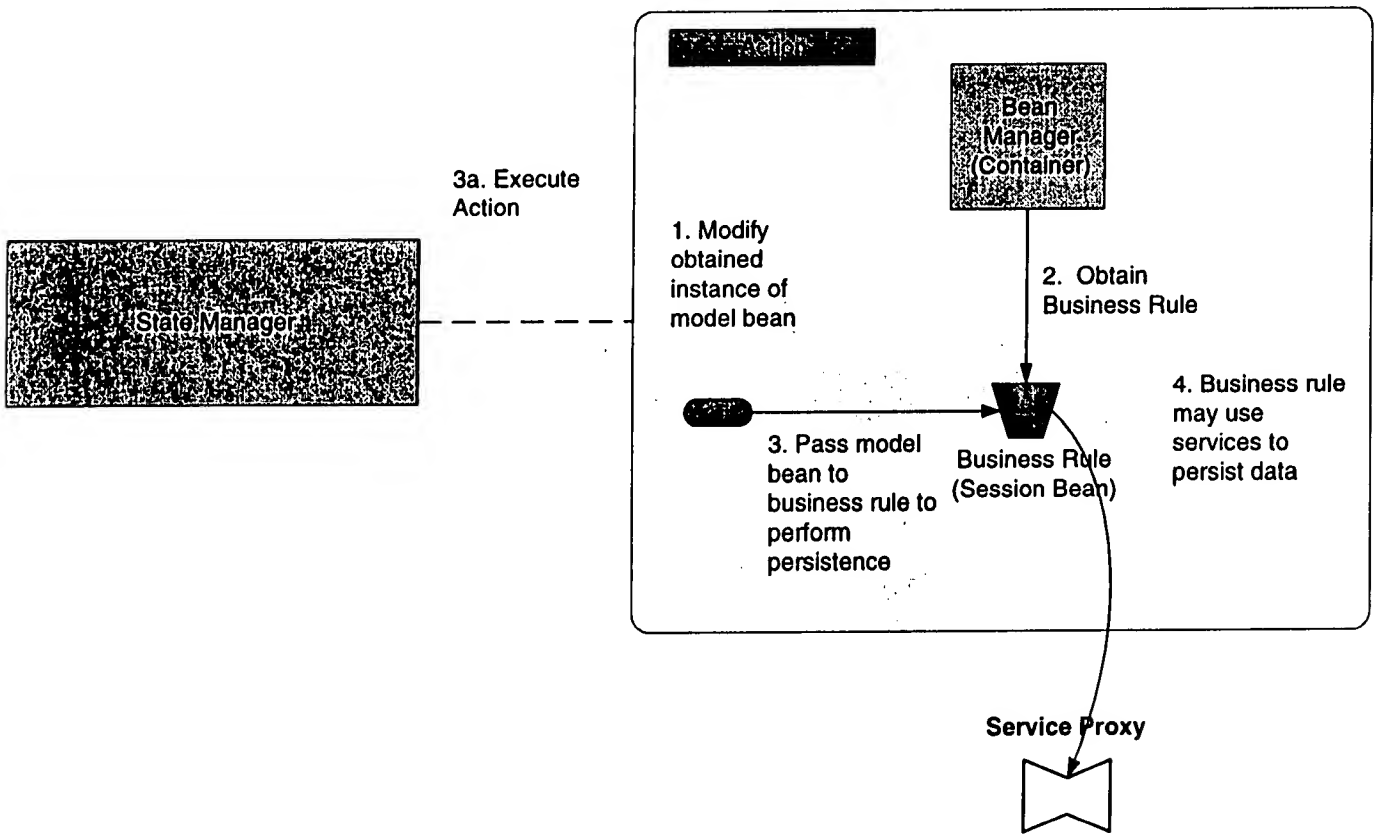
* Typically, an application that is in process will provide a requestContextID, which automatically identifies the application, ksd, and state to the state manager. The ksd also contains a default state so that state does not need to be specified. The request context ID and event name are the keys which allow the state manager to identify the next state to execute or display.

# Processing the Request




The State Servlet creates a Request Context Object from the servlet request, including the event, and passes that to the State Manager, which determines the appropriate state to run. If it is an action state, an action component is executed. If it is a display state, control is forwarded to the appropriate display (ksd).

# Performing An Action

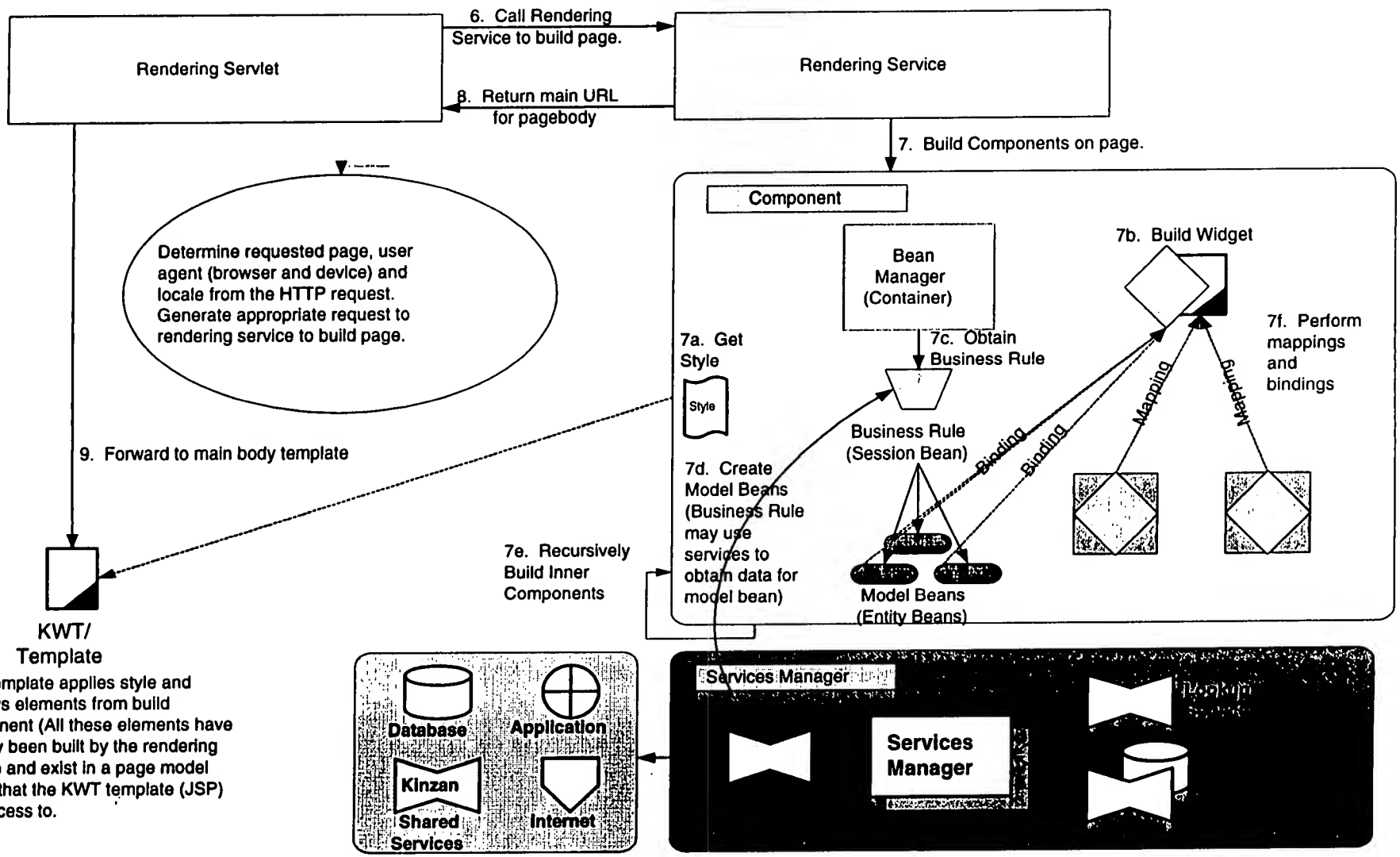



If the action state involves modifying the model, the action will have access to the model bean that represents the data to be modified. The data is then persisted through the business rule.

# State Diagram Detail

```
<ktp:wizard name="LoginWizard" startState="Login" directAccess="false" visibility="public">

    <ktp:displayState name="State1" kpd="KPD1" directAccess="true" >
        <ktp:transition event="Event1"  state="State2" />
        <ktp:transition event="Event2" state="State3" />
    </ktp:displayState>

    <ktp:actionState name="State2" component="net.kinzan.component.Component1">
          <ktp:transition event="Event3" state="State3" />
    </ktp:actionState>

    <ktp:displayState name="State3" kpd="KPD2" />

    <ktp:actionState name="failure" component="net.kinzan.component.Component1">
        <ktp:transition event="Event4"  state="State3" />
        <ktp:transition event="Event5" state="failure2" />
    </ktp:actionState>

    <ktp:displayState name="failure2" ksp="defaults/failure"/>

</ktp:wizard>
```

# Rendering the Component



# Rendering the Component: The Widget

```
<ktp:widget name="sampleWidget">
          <!-- The attribute list defines bindable attributes in the widget. A class
               is automatically be generated from the attribute list. -->
          <ktp:attributeList>
              <ktp:local name="text" type="java.util.String"/>
          </ktp:attributeList>

          <!-- style, device, and locale attributes default true and indicate
               to the preprocessor whether such variants should be searched for -->
          <ktp:widgetTemplate>sampleTemplate.kwt</widgetTemplate>
</ktp:widget>
```

- The widget attributes are **bindable** elements that are used by the template for display purposes.

The template may have variations based on style, locale, and/or device.

# Rendering the Component: The Widget Template

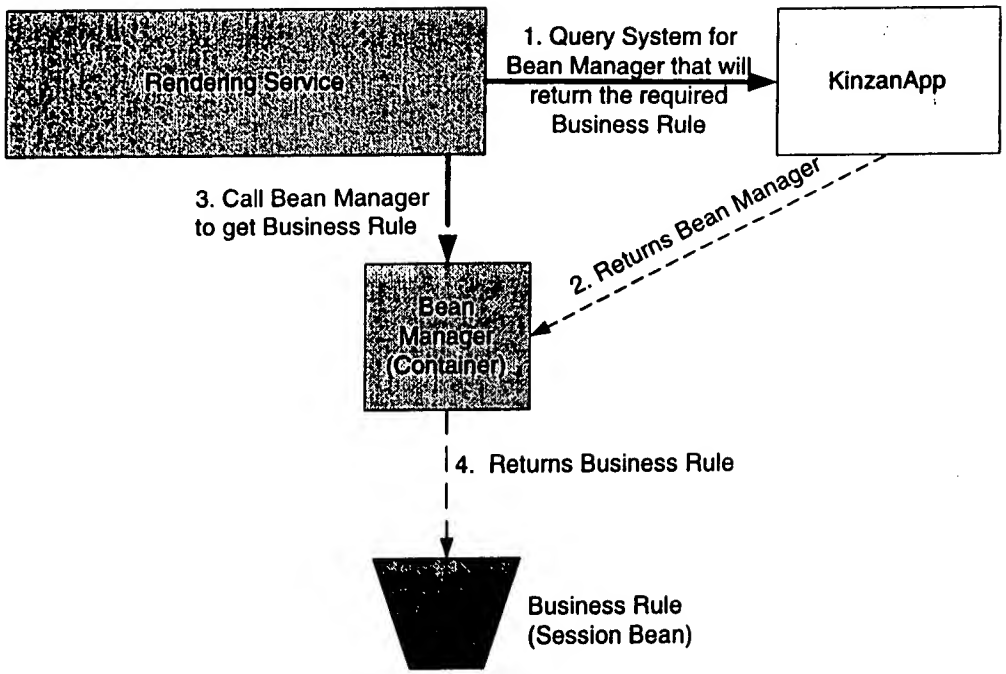
```
<!- Sample KWT file for simple widget example -->

<table>
    <tr><td><ktp:zone name="zone1"/></td></tr>

    <tr><td><ktp:attribute name="text"/></td></tr>

</table>
```

**Zone** is a placeholder for other components to be **mapped** into

**Attribute** places the value of the named attribute from the widget into the template.

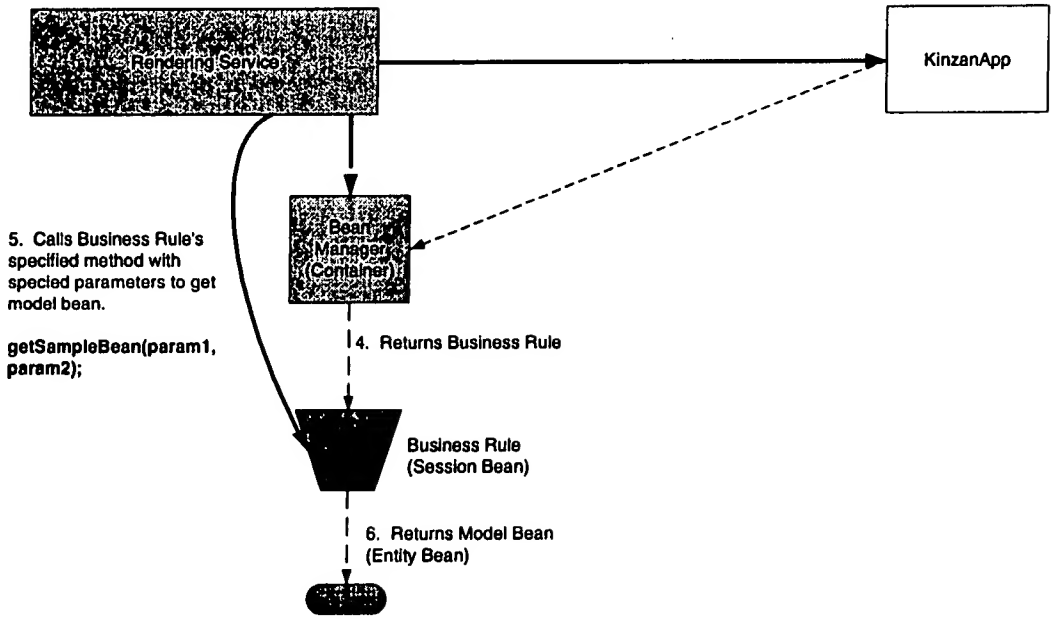
The loader processes the KWT into a JSP containing the appropriate JSP code to perform the required mappings.

# Rendering the Component: The Bean Manager




```
<ktp:beanManager name="SampleBeanManager">
      <ktp:beanClass>net.kinzan.model.beanmanager.SampleBeanManager</ktp:beanClass>
      <ktp:propertyFile>
            http://codebase.kinzan.com/codebase.SampleBeanManager.properties
      </ktp:propertyFile>
</ktp:beanManager>
```

# Rendering the Component: The Business Rule (1)

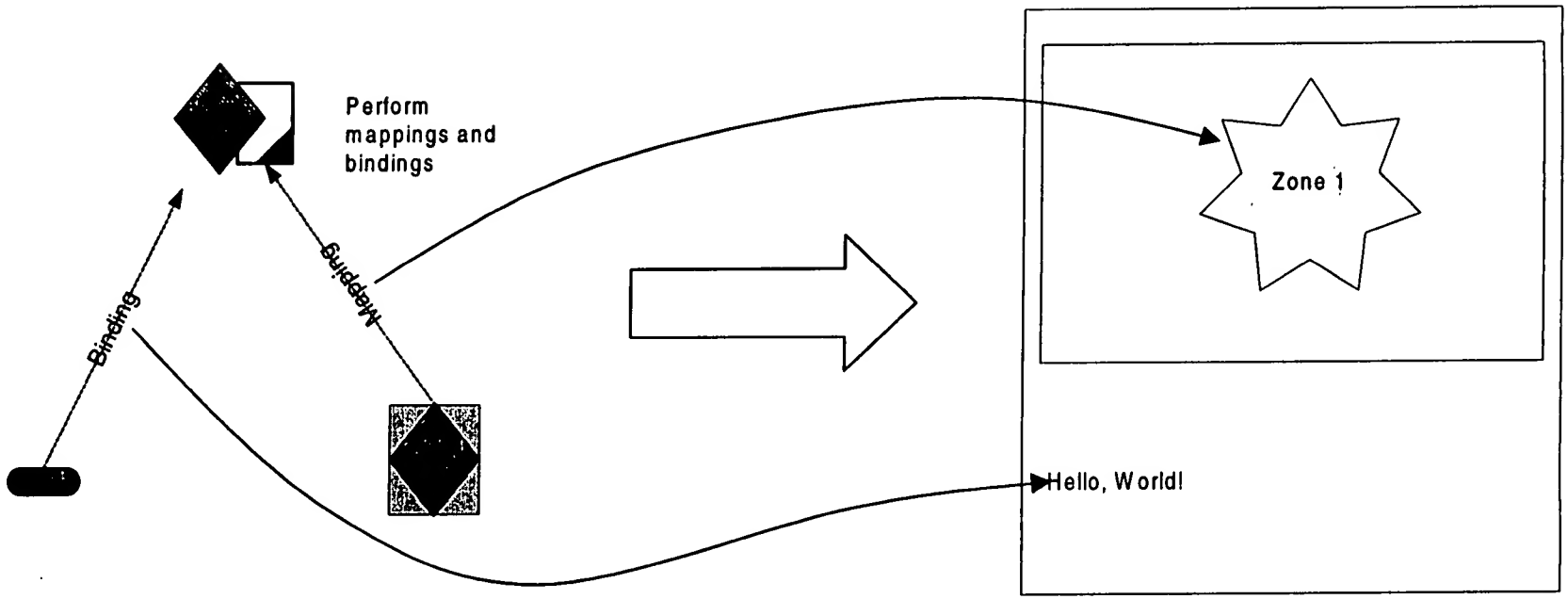



```
<ktp:modelBean name="sampleBean">
        <ktp:rule name="sampleBusinessRule" method="getSampleBean"/>
        <ktp:param name="param1" type="java.lang.String">
                <ktp:from widget="sampleWidget" attribute="text"/>
        </ktp:param>
        <ktp:param name="param2" type="java.lang.String">
            <ktp:from literal="param2"/>
        </ktp:param>
</ktp:modelBean>
```

# Rendering the Component: The Business Rule (2)

```
public class SampleBusinessRule extends KinzanBusinessRule
{
        ...
        public SampleBean getSampleBean( java.lang.String param1, java.lang.String param2 )
        {
                InitialContext ic = new InitialContext();

                // Get home interface
                SampleBeanHome sbh = (SampleBeanHome)ic.lookup("SampleBeanHome");

                // Get remote interface to entity bean.  param1, param2 form the primary key.
                SampleBeanPK pk = new SampleBeanPK();
                pk.key1 = param1;
                pk.key2 = param2;

                SampleBean sb = sbh.findByPrimaryKey( pk );

                return sb;
        }
        ...
}
```

# Rendering the Component: Bindings & Mappings




```
...
    <ktp:binding widget="SampleWidget" attribute="text">
        <ktp:from modelBean="sampleBean" attribute="message"/>
    </ktp:binding>
...
<ktp:componentModeList>
    <ktp:componentMode widget="sampleWidget">
        <ktp:zone name="zone1" component="anotherComponent"/>
    </ktp:componentMode>
</ktp:componentModeList>
```

In the foregoing, the method and apparatus of the present invention is described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the present invention. In particular, the separate blocks of the various block diagrams represent functional blocks of methods or apparatuses and are not necessarily indicative of physical or logical separations or of an order of operation inherent in the spirit and scope of the present invention. For example, the various blocks of some figures may be integrated into components, or may be subdivided into components. Moreover, the various blocks of other figures represent portions of a method which, in some embodiments, may be reordered or may be organized in a parallel or a linear or step-wise fashion. The present specification and figures are accordingly to be regarded as illustrative rather than restrictive.

## CLAIMS

What is claimed is that which has been described in the foregoing and equivalents thereof.